FIG. 1

(a)

```
int a[];
for (i=0; i<128; i++){
      x += a[i];
}
                        .
                        .
                        .
```

(b)

```
int a[];
for (i=0; i<128; i++){
      dpref(&a[i + N]);    // Prefetch data a certain number (N, in this case) of iterations ahead
      x += a[i];           // in consideration of latency caused until reference is made
}
                        .
                        .
                        .
```

FIG. 2

```
for (i = 0; i < 128; ) {
        dpref(&a[i+32]);
        for (j = 0; j < 32; j++, i++) {
                x += a[i];
        }
}
```

FIG. 3

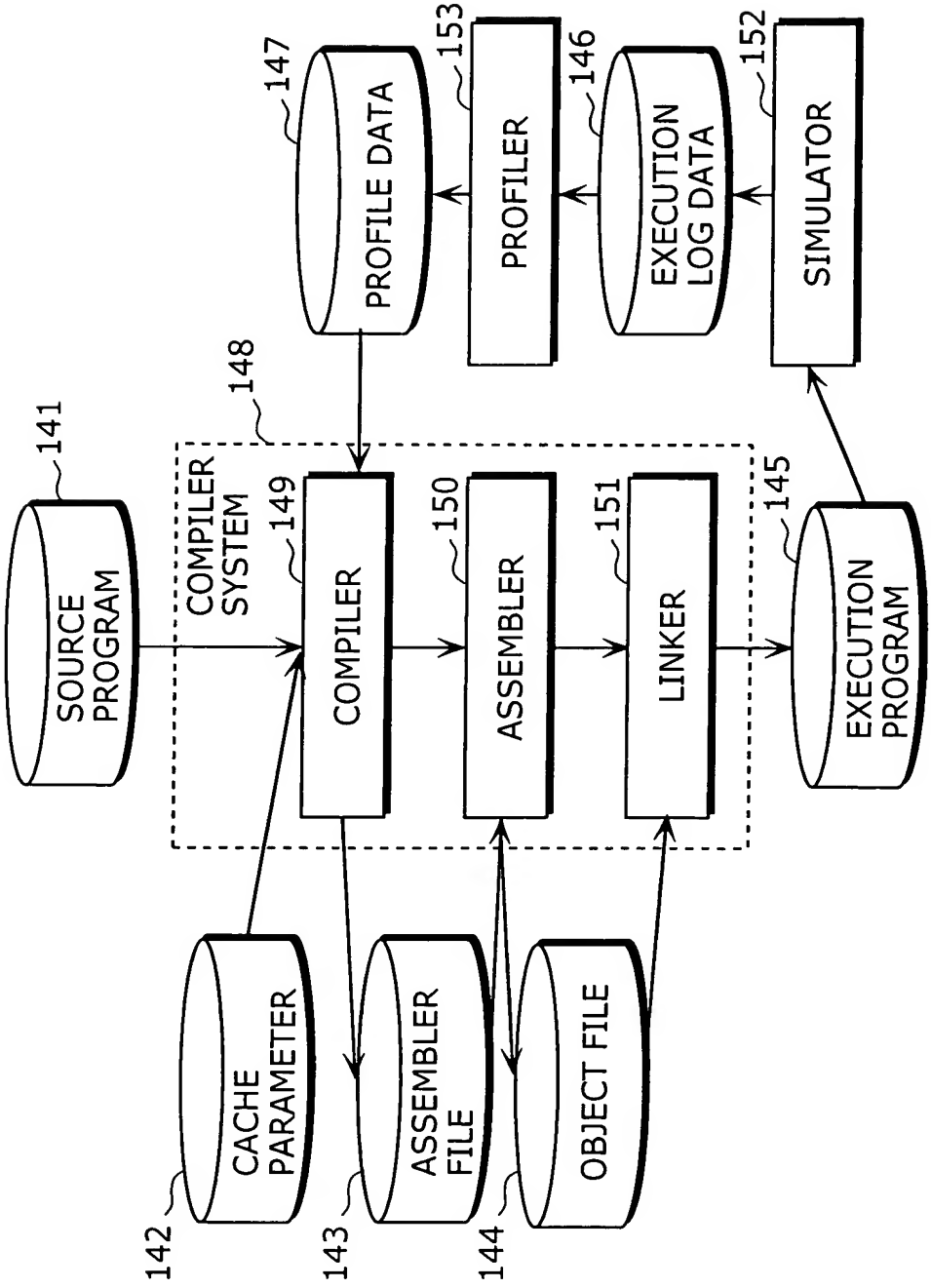

SOURCE PROGRAM
141
COMPILER SYSTEM
148
COMPILER
149
ASSEMBLER
150
LINKER
151
EXECUTION PROGRAM
145
PROFILE DATA
147
PROFILER
153
EXECUTION LOG DATA
146
SIMULATOR
152
CACHE PARAMETER
142
ASSEMBLER FILE
143
OBJECT FILE
144

FIG. 4

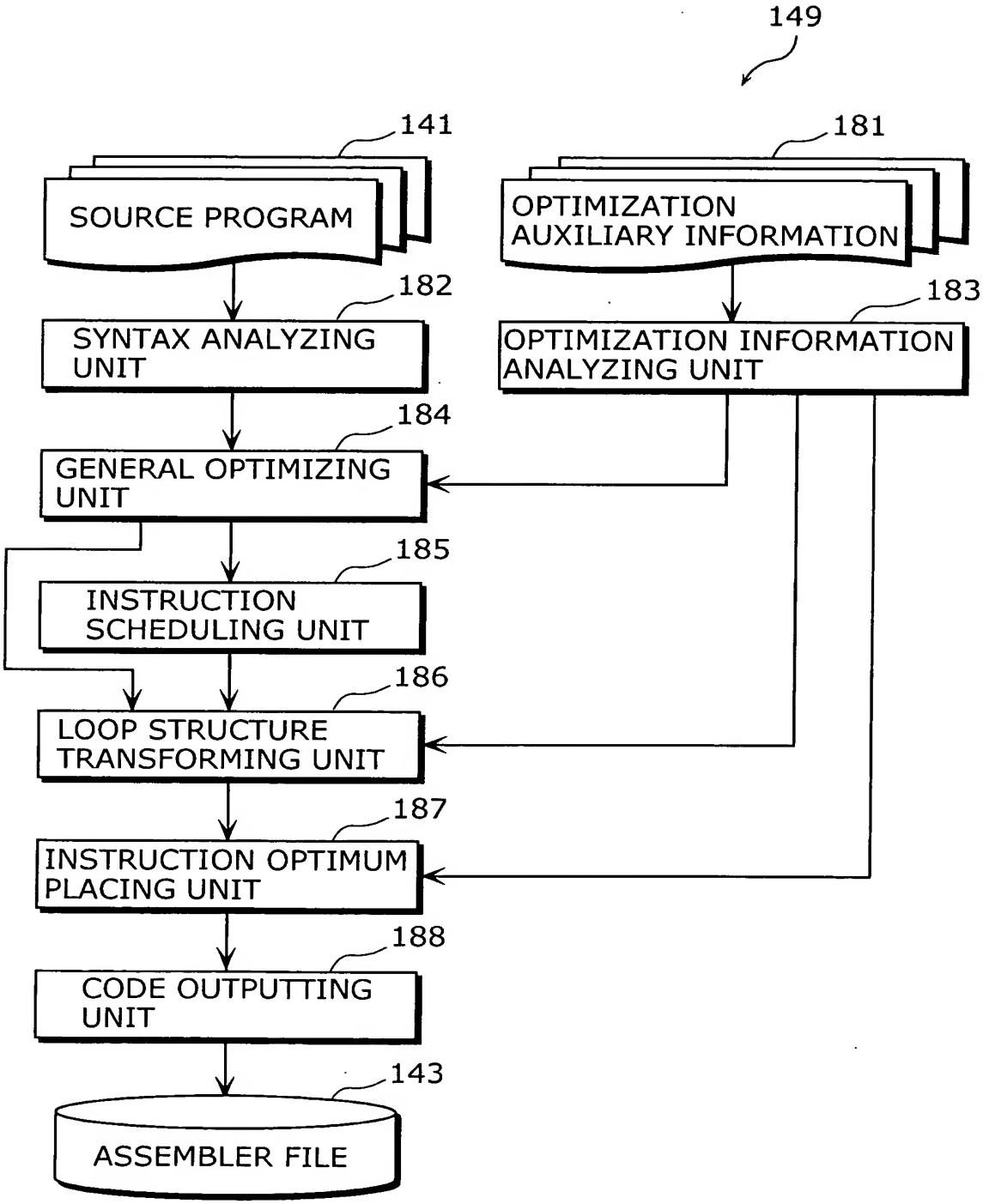

149
141
SOURCE PROGRAM
181
OPTIMIZATION AUXILIARY INFORMATION
182
SYNTAX ANALYZING UNIT
183
OPTIMIZATION INFORMATION ANALYZING UNIT
184
GENERAL OPTIMIZING UNIT
185
INSTRUCTION SCHEDULING UNIT
186
LOOP STRUCTURE TRANSFORMING UNIT
187
INSTRUCTION OPTIMUM PLACING UNIT
188
CODE OUTPUTTING UNIT
143
ASSEMBLER FILE

FIG. 5

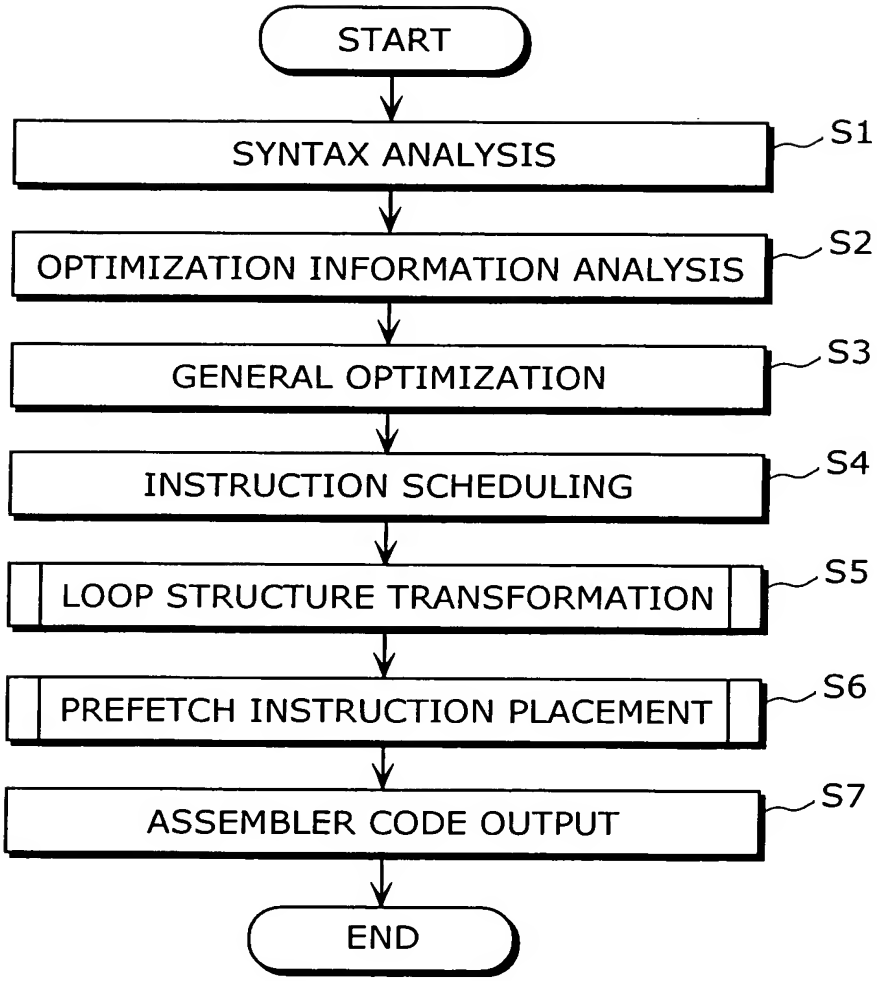

START
SYNTAX ANALYSIS
S1
OPTIMIZATION INFORMATION ANALYSIS
S2
GENERAL OPTIMIZATION
S3
INSTRUCTION SCHEDULING
S4
LOOP STRUCTURE TRANSFORMATION
S5
PREFETCH INSTRUCTION PLACEMENT
S6
ASSEMBLER CODE OUTPUT
S7
END

FIG. 6

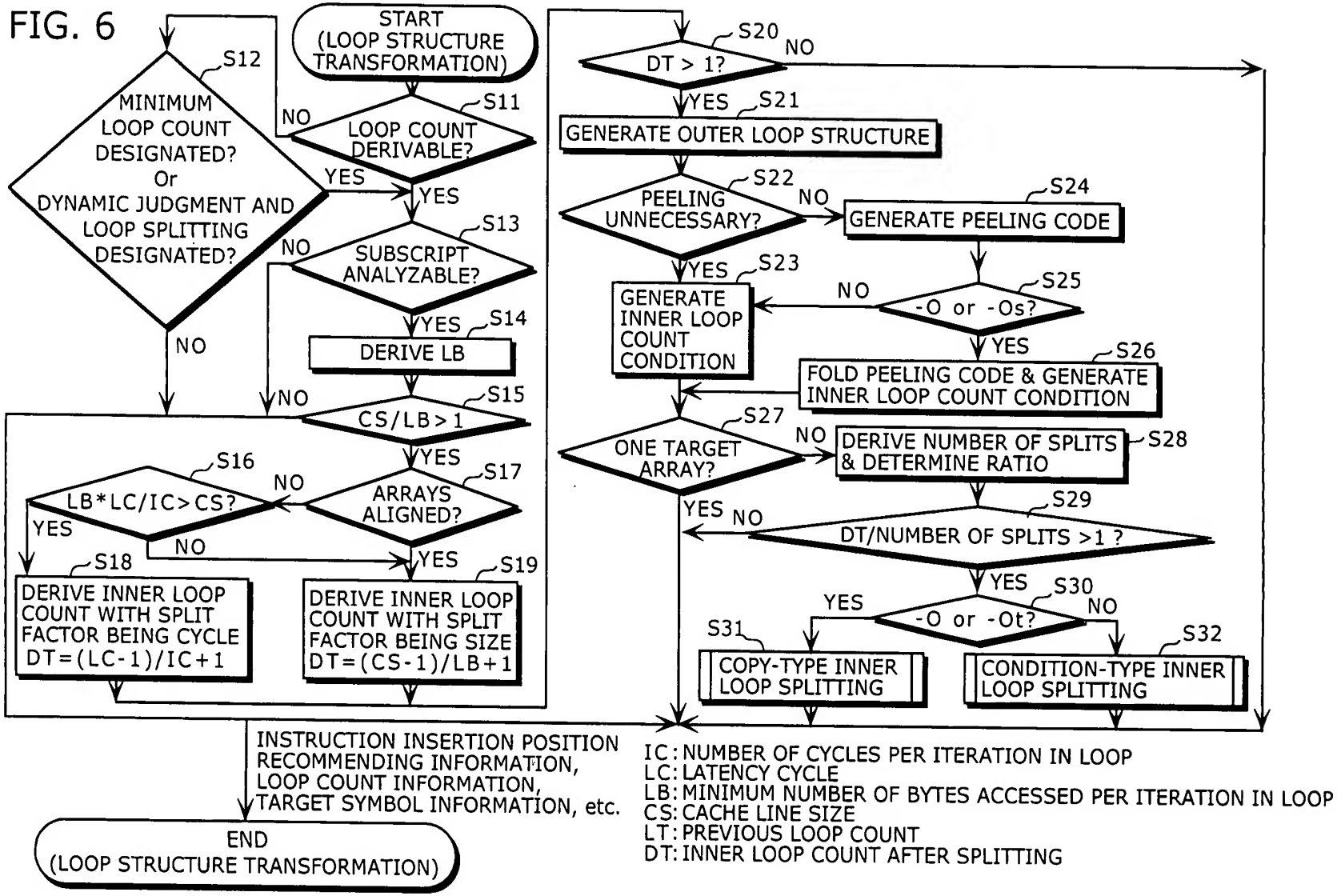

START (LOOP STRUCTURE TRANSFORMATION)
S11
LOOP COUNT DERIVABLE?
NO
YES
S12
MINIMUM LOOP COUNT DESIGNATED? Or DYNAMIC JUDGMENT AND LOOP SPLITTING DESIGNATED?
YES
NO
S13
SUBSCRIPT ANALYZABLE?
NO
YES
S14
DERIVE LB
S15
CS/LB > 1
NO
YES
S16
LB*LC/IC > CS?
YES
NO
S17
ARRAYS ALIGNED?
NO
YES
S18
DERIVE INNER LOOP COUNT WITH SPLIT FACTOR BEING CYCLE DT=(LC-1)/IC+1
S19
DERIVE INNER LOOP COUNT WITH SPLIT FACTOR BEING SIZE DT=(CS-1)/LB+1
S20
DT > 1?
NO
YES
S21
GENERATE OUTER LOOP STRUCTURE
S22
PEELING UNNECESSARY?
NO
YES
S24
GENERATE PEELING CODE
S25
-O or -Os?
NO
YES
S23
GENERATE INNER LOOP COUNT CONDITION
S26
FOLD PEELING CODE & GENERATE INNER LOOP COUNT CONDITION
S27
ONE TARGET ARRAY?
NO
YES
S28
DERIVE NUMBER OF SPLITS & DETERMINE RATIO
S29
DT/NUMBER OF SPLITS > 1 ?
NO
YES
S30
-O or -Ot?
YES
NO
S31
COPY-TYPE INNER LOOP SPLITTING
S32
CONDITION-TYPE INNER LOOP SPLITTING
INSTRUCTION INSERTION POSITION RECOMMENDING INFORMATION, LOOP COUNT INFORMATION, TARGET SYMBOL INFORMATION, etc.
END (LOOP STRUCTURE TRANSFORMATION)
IC: NUMBER OF CYCLES PER ITERATION IN LOOP
LC: LATENCY CYCLE
LB: MINIMUM NUMBER OF BYTES ACCESSED PER ITERATION IN LOOP
CS: CACHE LINE SIZE
LT: PREVIOUS LOOP COUNT
DT: INNER LOOP COUNT AFTER SPLITTING

FIG. 7

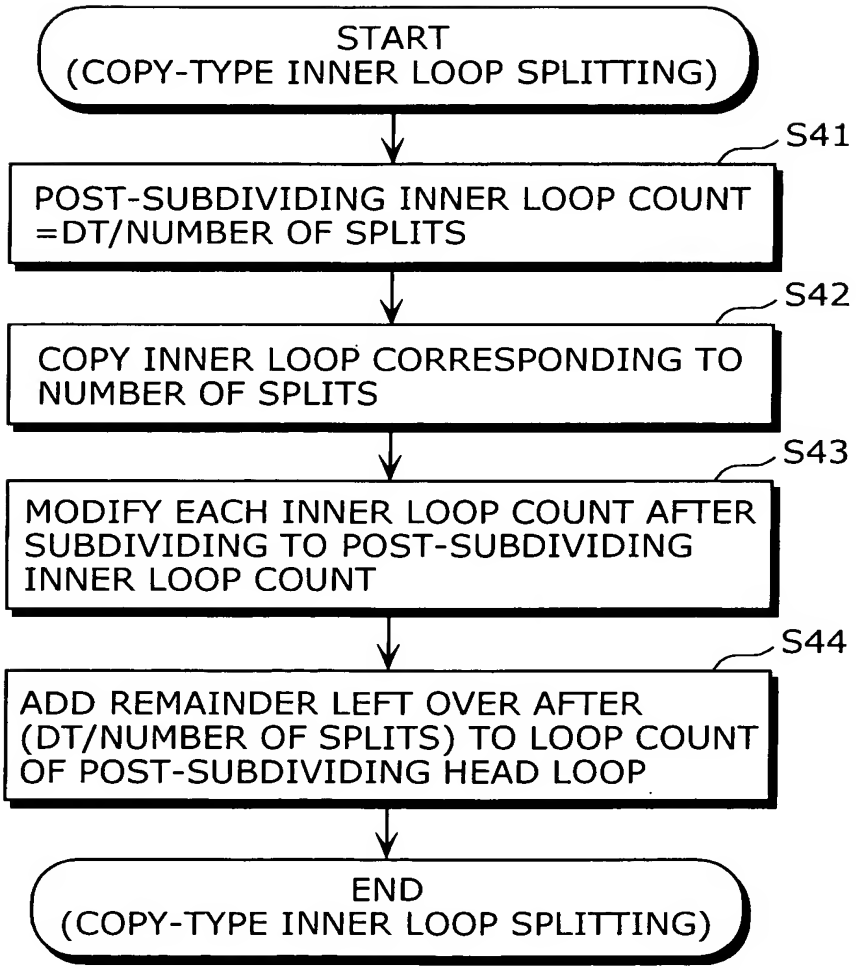

START
(COPY-TYPE INNER LOOP SPLITTING)
S41
POST-SUBDIVIDING INNER LOOP COUNT
=DT/NUMBER OF SPLITS
S42
COPY INNER LOOP CORRESPONDING TO
NUMBER OF SPLITS
S43
MODIFY EACH INNER LOOP COUNT AFTER
SUBDIVIDING TO POST-SUBDIVIDING
INNER LOOP COUNT
S44
ADD REMAINDER LEFT OVER AFTER
(DT/NUMBER OF SPLITS) TO LOOP COUNT
OF POST-SUBDIVIDING HEAD LOOP
END
(COPY-TYPE INNER LOOP SPLITTING)

FIG. 8

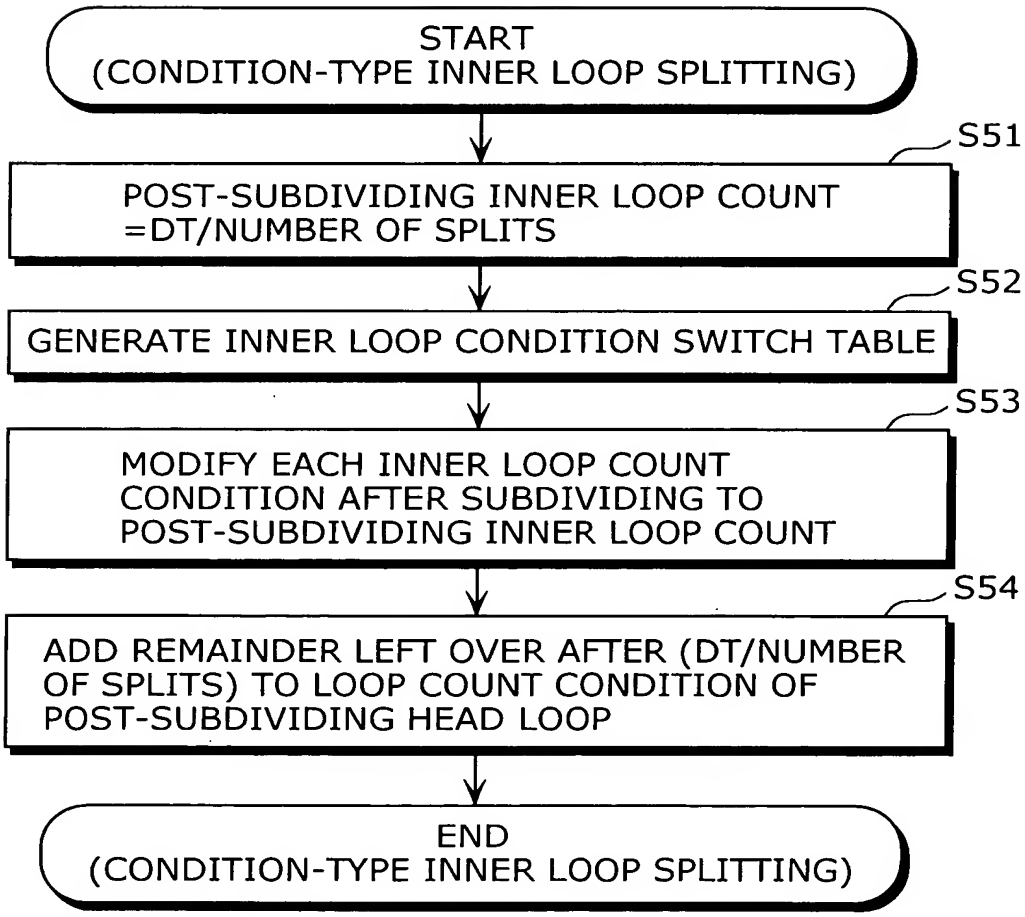

START
(CONDITION-TYPE INNER LOOP SPLITTING)
S51
POST-SUBDIVIDING INNER LOOP COUNT
=DT/NUMBER OF SPLITS
S52
GENERATE INNER LOOP CONDITION SWITCH TABLE
S53
MODIFY EACH INNER LOOP COUNT
CONDITION AFTER SUBDIVIDING TO
POST-SUBDIVIDING INNER LOOP COUNT
S54
ADD REMAINDER LEFT OVER AFTER (DT/NUMBER
OF SPLITS) TO LOOP COUNT CONDITION OF
POST-SUBDIVIDING HEAD LOOP
END
(CONDITION-TYPE INNER LOOP SPLITTING)

FIG. 9

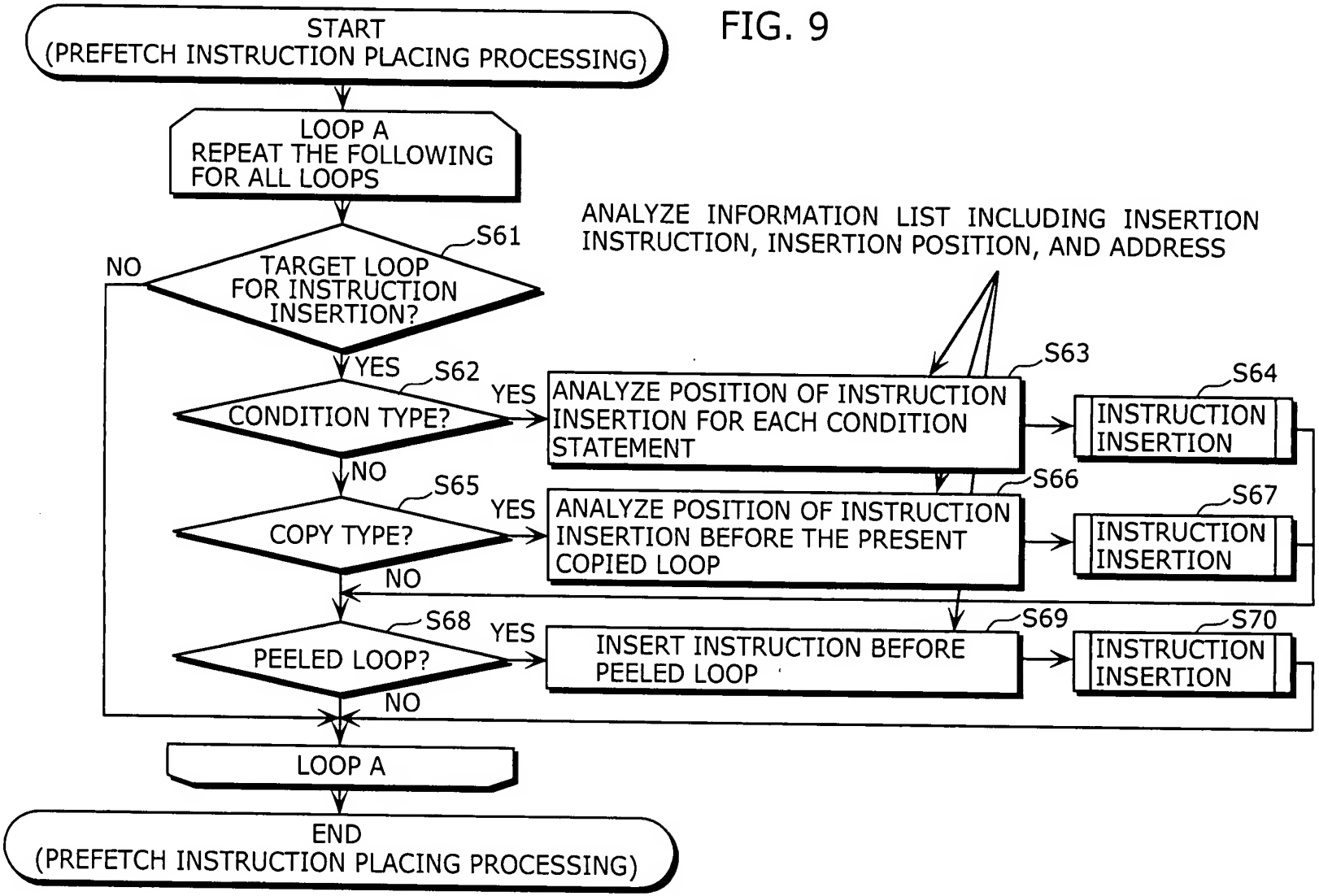

START
(PREFETCH INSTRUCTION PLACING PROCESSING)
LOOP A
REPEAT THE FOLLOWING
FOR ALL LOOPS
S61
TARGET LOOP
FOR INSTRUCTION
INSERTION?
NO
YES
S62
CONDITION TYPE?
YES
NO
S65
COPY TYPE?
YES
NO
S68
PEELED LOOP?
YES
NO
LOOP A
END
(PREFETCH INSTRUCTION PLACING PROCESSING)
ANALYZE INFORMATION LIST INCLUDING INSERTION
INSTRUCTION, INSERTION POSITION, AND ADDRESS
S63
ANALYZE POSITION OF INSTRUCTION
INSERTION FOR EACH CONDITION
STATEMENT
S64
INSTRUCTION
INSERTION
S66
ANALYZE POSITION OF INSTRUCTION
INSERTION BEFORE THE PRESENT
COPIED LOOP
S67
INSTRUCTION
INSERTION
S69
INSERT INSTRUCTION BEFORE
PEELED LOOP
S70
INSTRUCTION
INSERTION

FIG. 10

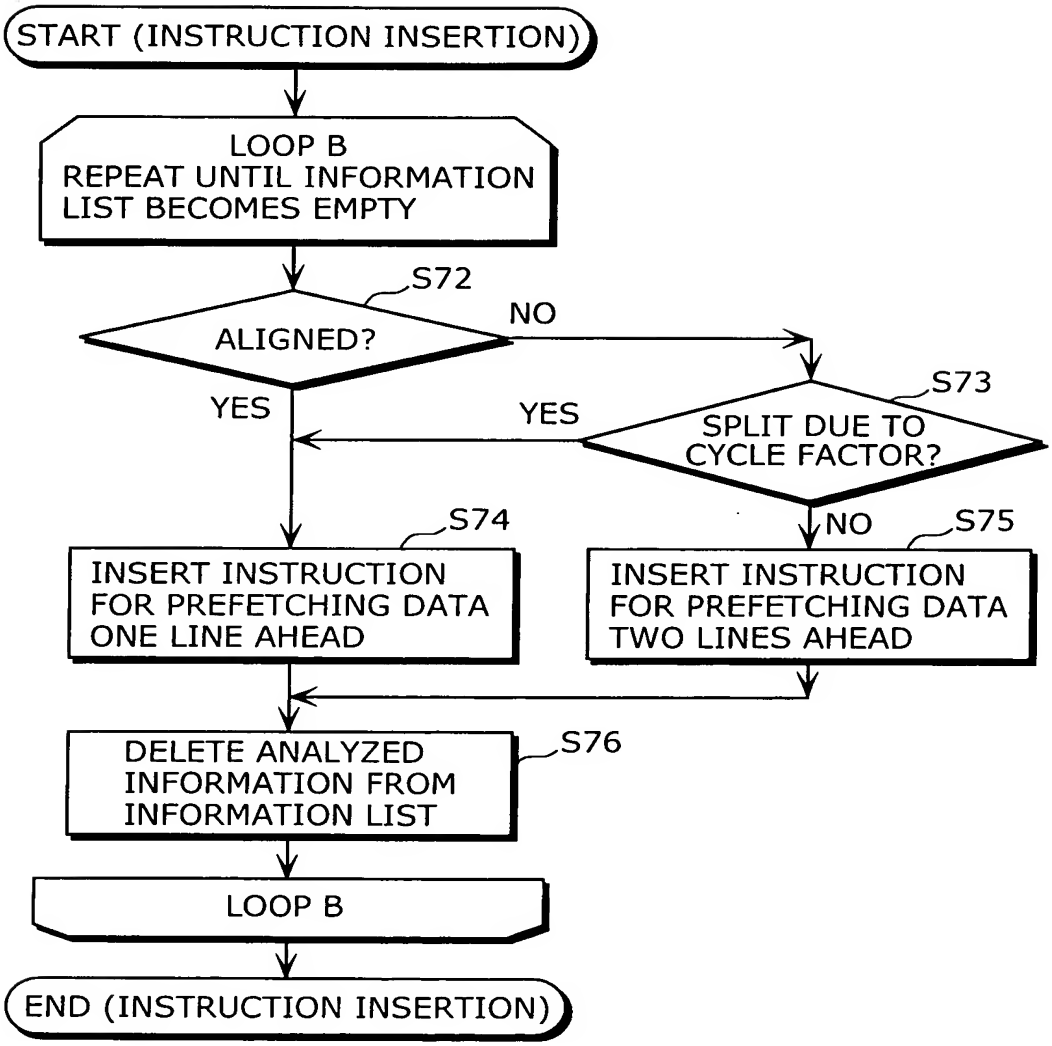

START (INSTRUCTION INSERTION)
LOOP B
REPEAT UNTIL INFORMATION LIST BECOMES EMPTY
S72
ALIGNED?
NO
YES
S73
SPLIT DUE TO CYCLE FACTOR?
YES
NO
S74
INSERT INSTRUCTION FOR PREFETCHING DATA ONE LINE AHEAD
S75
INSERT INSTRUCTION FOR PREFETCHING DATA TWO LINES AHEAD
S76
DELETE ANALYZED INFORMATION FROM INFORMATION LIST
LOOP B
END (INSTRUCTION INSERTION)

FIG. 11

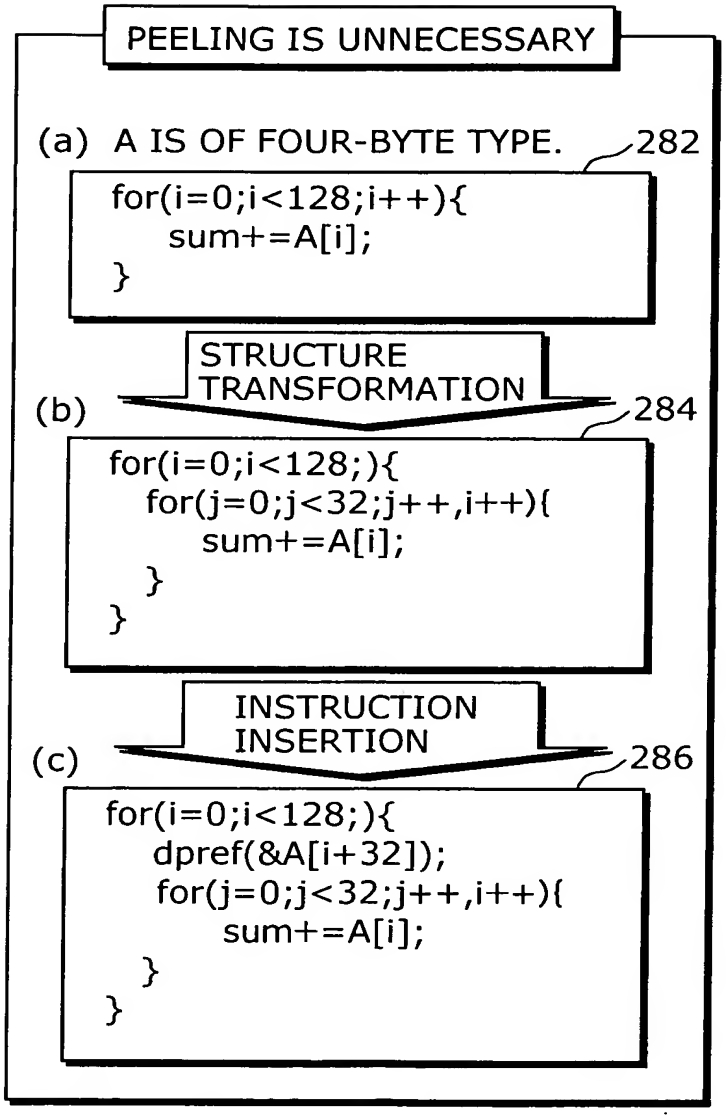

PEELING IS UNNECESSARY
(a) A IS OF FOUR-BYTE TYPE.
282
for(i=0;i<128;i++){
sum+=A[i];
}
STRUCTURE TRANSFORMATION
(b)
284
for(i=0;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
INSTRUCTION INSERTION
(c)
286
for(i=0;i<128;){
dpref(&A[i+32]);
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}

FIG. 12

INPUT PROGRAM SOURCE IN C LANGUAGE

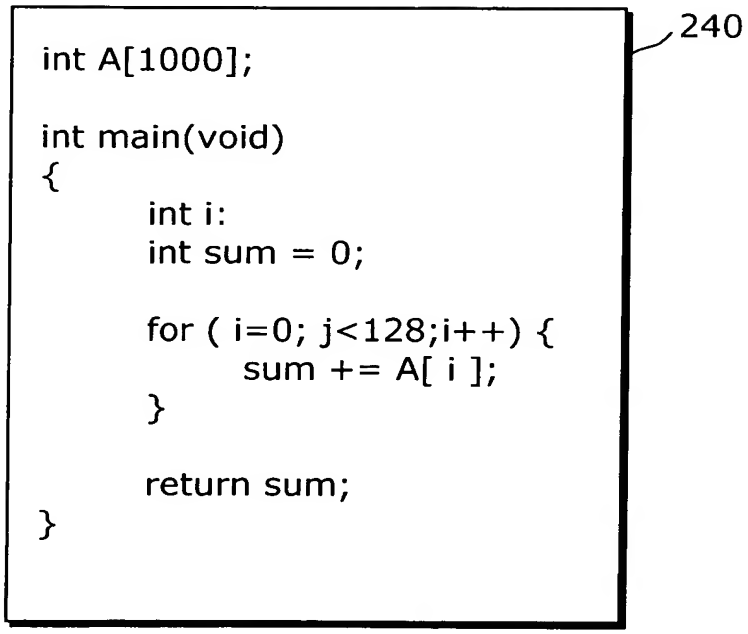


FIG. 13 INTERMEDIATE LANGUAGE INPUT BY TRANSFORMING UNIT

```
[PROLOG]
[BGNBBLK] B1      [predecess set] no          [success set] B2
          mov     REG (vr2) | IMM(0)
          mov     REG (vr5) | IMM(128)
          ld      REG (vr3) | IMM(_A$)
          mov     REG (vr1) | IMM(vr2)

[ENDBBLK]



[BGNBBLK] B2      [predecess set] B1 B2       [success set] B2 B3
[label] L00001
          add     REG (vr2) | REG (vr2), IMM(1)
          ldinc   REG (vr4), REG(vr3) | INDIRECT(vr3,0), REG(vr3), IMM(4)
          cmplt   FLAG(C6) | REG (vr2), REG (vr5)
          add     REG (vr1) | REG (vr4), REG (vr1)
          jmpf    | FLAG (C6), LAB(L00001)
[ENDBBLK]



[BGNBBLK] B3      [predecess set] B2          [success set] no
          mov     REG (r0) | REG (vr1)
          ret
[ENDBBLK]
[EPILOG]
```

FIG. 14 INTERMEDIATE LANGUAGE AFTER TRANSFORMATION

```
[PROLOG]
[BGNBBLK] B1        [predecess set] no          [success set] B4
                    mov             REG (vr2) | IMM(0)
                    mov             REG (vr5) | IMM(32)
                    ld              REG (vr3) | MEM(_A$)
                    mov             REG (vr1) | REG(vr2)
                    mov             REG (vr7) | IMM(128)
[ENDBBLK]
[BGNBBLK] B4        [predecess set] B1 B4       [success set] B2
[label] L00002

[ENDBBLK]
[BGNBBLK] B2        [predecess set] B4 B2       [success set] B2 B5
[label] L00001
                    add             REG (vr2) | REG (vr2), IMM(1)
                    ldinc           REG (vr4), REG(vr3) | INDIRECT(vr3,0), REG(vr3), IMM(4)
                    cmplt           FLAG(C6) | REG (vr2), REG (vr5)
                    add             REG (vr1) | REG (vr4), REG (vr1)
                    jmpf             | FLAG (C6), LAB(L00001)
[ENDBBLK]
[BGNBBLK] B5        [predecess set] B2          [success set] B5 B3
                    cmplt           FLAG (C6) | REG (vr2), REG (vr7)
                    jmpf             | FLAG (C6), LAB(L00002)
[ENDBBLK]
[BGNBBLK] B3        [predecess set] B5          [success set] no
                    mov             REG (r0) | REG (vr1)
                    ret
[ENDBBLK]
[EPILOG]
```

FIG. 15 INTERMEDIATE LANGUAGE AFTER INSTRUCTION INSERTION

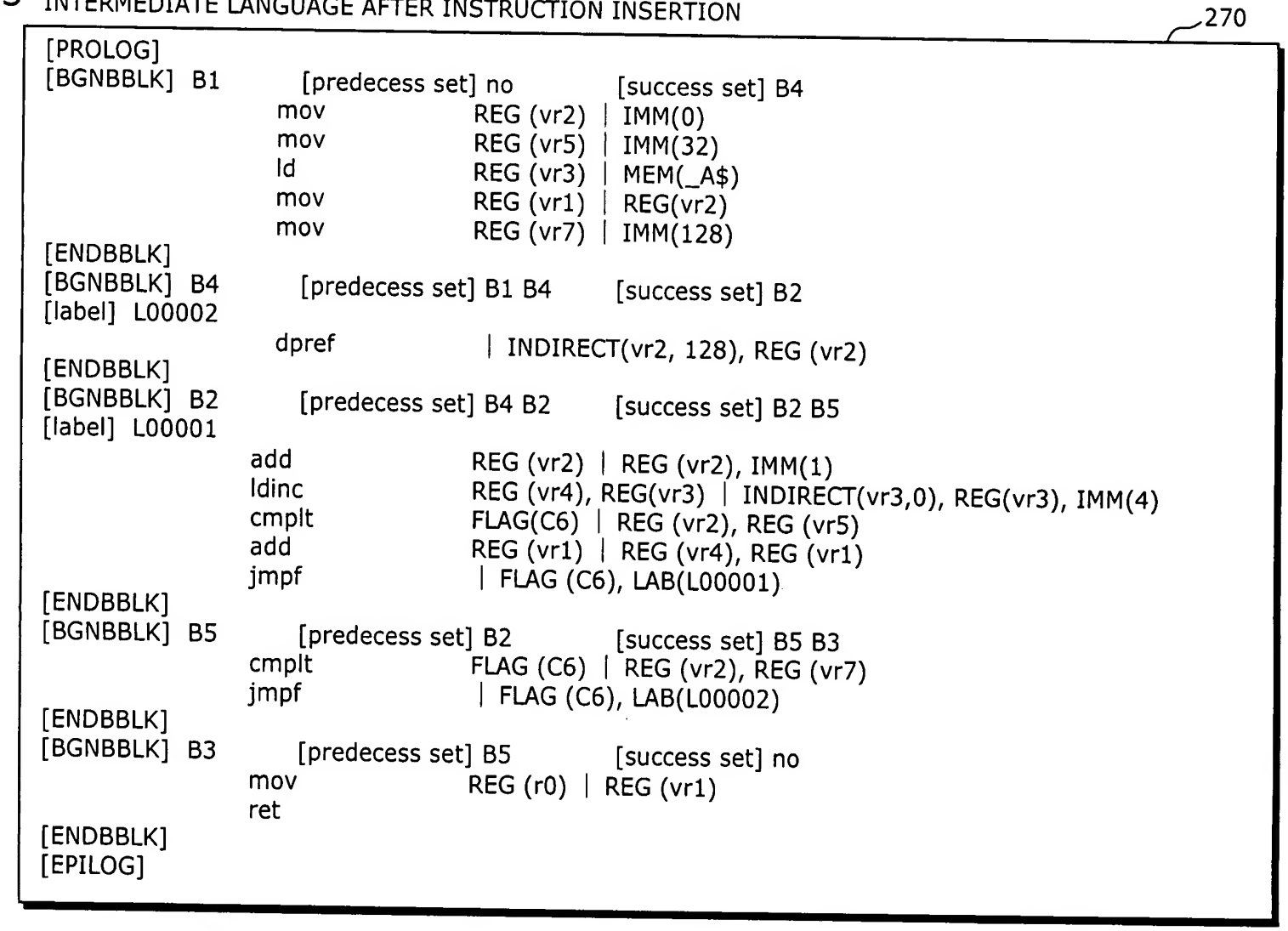


FIG. 16

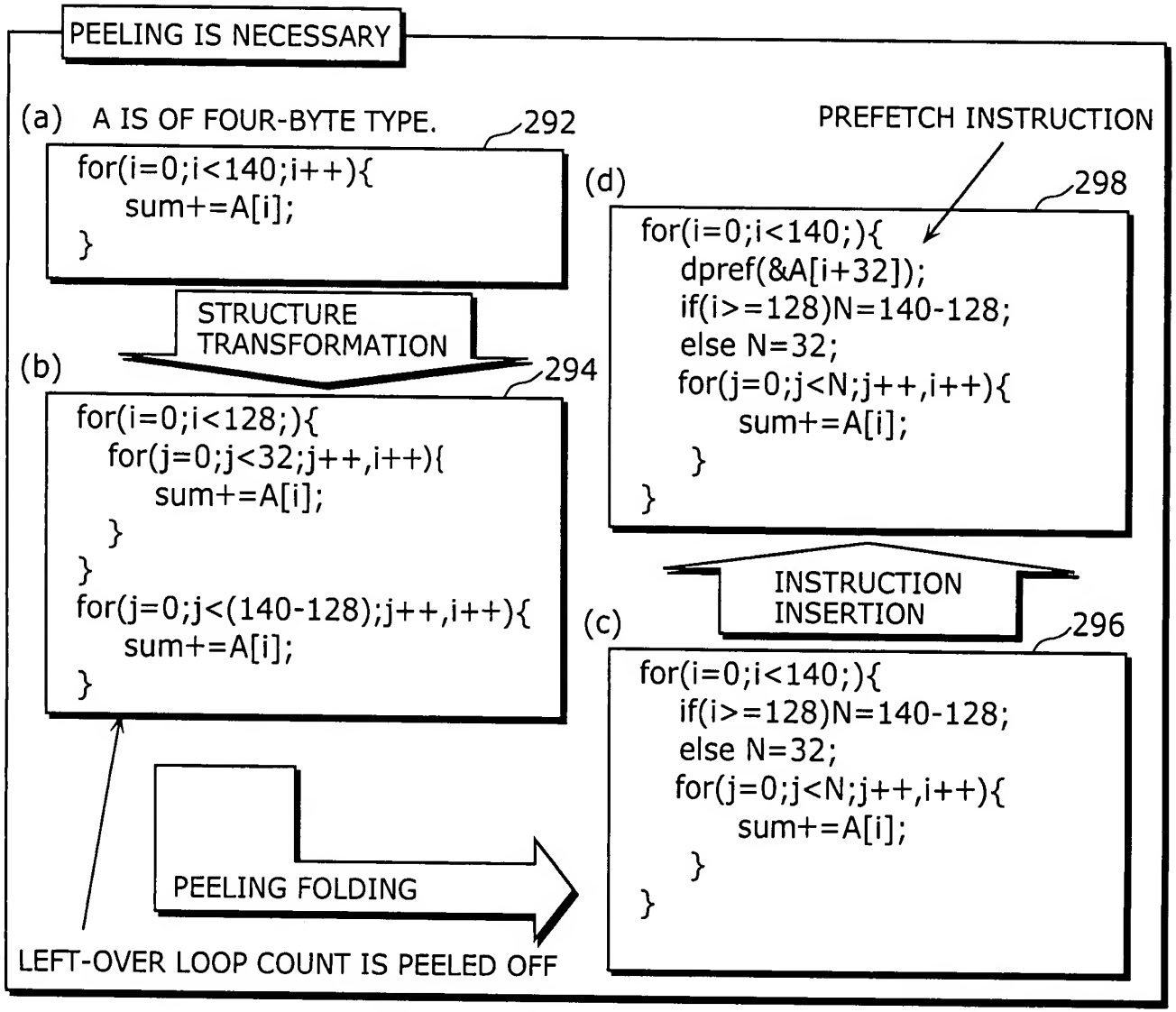

PEELING IS NECESSARY
(a) A IS OF FOUR-BYTE TYPE.
292
for(i=0;i<140;i++){
sum+=A[i];
}
STRUCTURE TRANSFORMATION
(b)
294
for(i=0;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
for(j=0;j<(140-128);j++,i++){
sum+=A[i];
}
PEELING FOLDING
LEFT-OVER LOOP COUNT IS PEELED OFF
PREFETCH INSTRUCTION
(d)
298
for(i=0;i<140;){
dpref(&A[i+32]);
if(i>=128)N=140-128;
else N=32;
for(j=0;j<N;j++,i++){
sum+=A[i];
}
}
INSTRUCTION INSERTION
(c)
296
for(i=0;i<140;){
if(i>=128)N=140-128;
else N=32;
for(j=0;j<N;j++,i++){
sum+=A[i];
}
}

FIG. 17

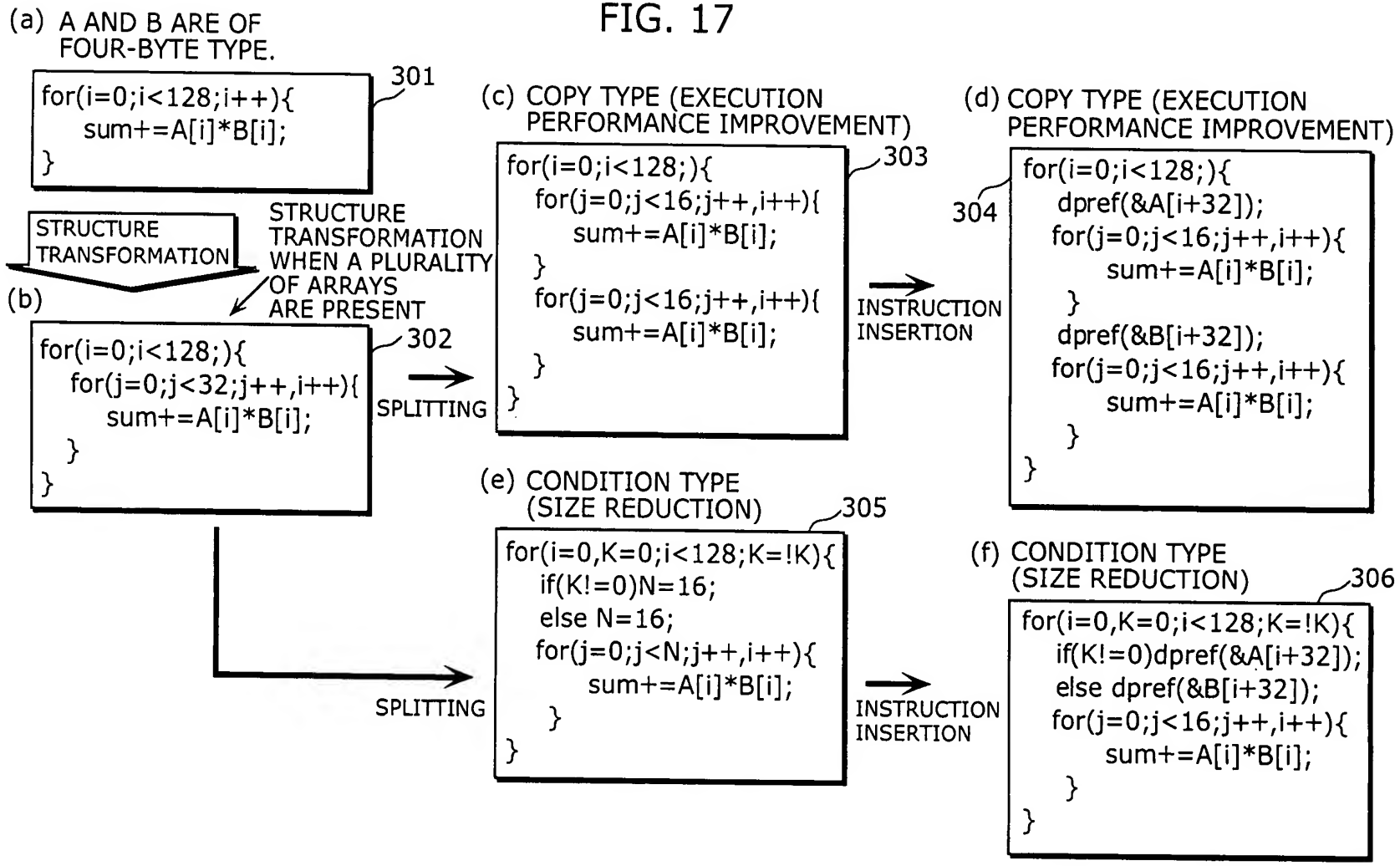

(a) A AND B ARE OF FOUR-BYTE TYPE.
for(i=0;i<128;i++){
sum+=A[i]*B[i];
}
301
STRUCTURE TRANSFORMATION
STRUCTURE TRANSFORMATION WHEN A PLURALITY OF ARRAYS ARE PRESENT
(b)
for(i=0;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i]*B[i];
}
}
302
SPLITTING
(c) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
for(i=0;i<128;){
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
}
303
INSTRUCTION INSERTION
(d) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
304
for(i=0;i<128;){
dpref(&A[i+32]);
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
dpref(&B[i+32]);
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
}
SPLITTING
(e) CONDITION TYPE (SIZE REDUCTION)
for(i=0,K=0;i<128;K=!K){
if(K!=0)N=16;
else N=16;
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}
305
INSTRUCTION INSERTION
(f) CONDITION TYPE (SIZE REDUCTION)
306
for(i=0,K=0;i<128;K=!K){
if(K!=0)dpref(&A[i+32]);
else dpref(&B[i+32]);
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
}

FIG. 18

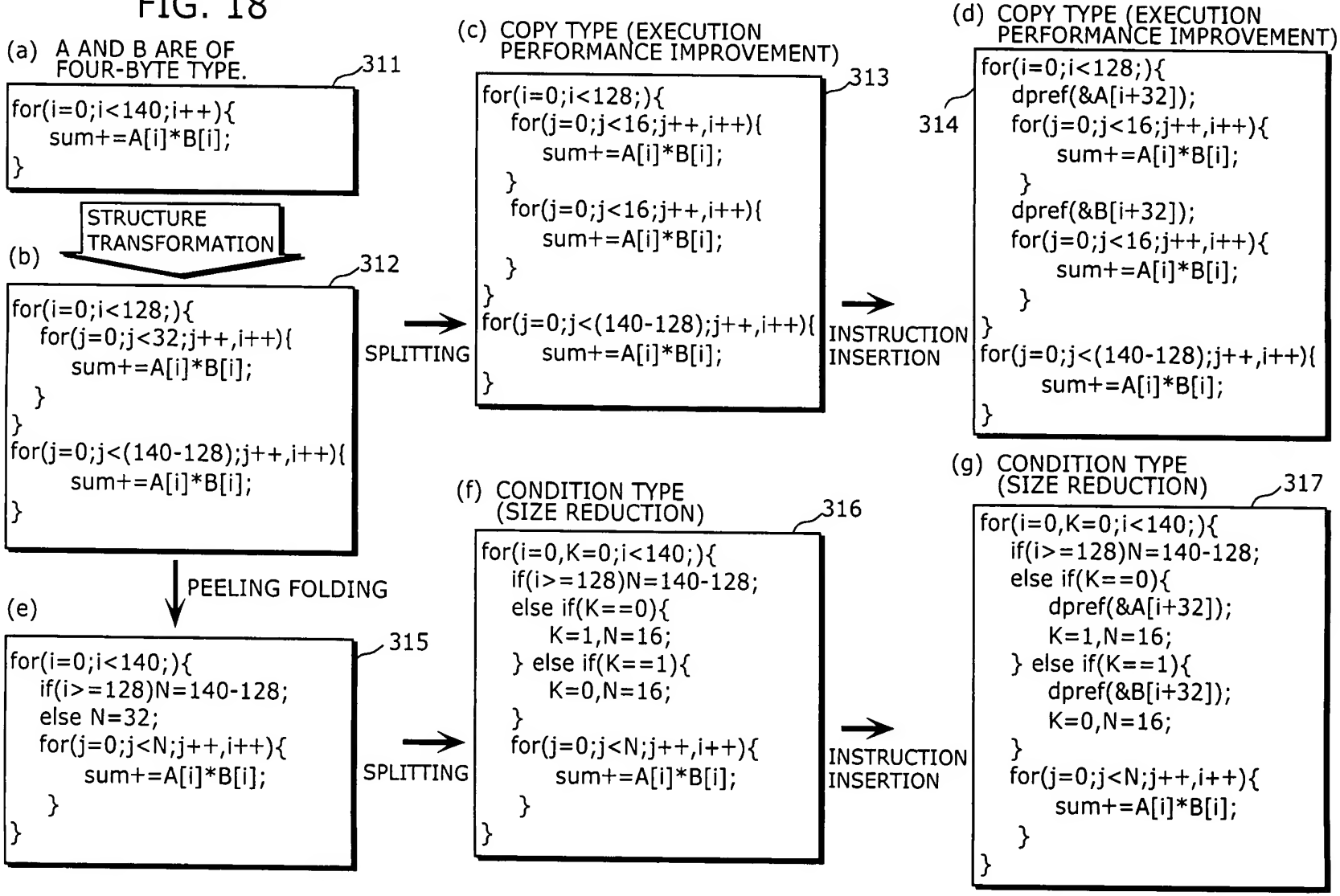

(a) A AND B ARE OF FOUR-BYTE TYPE.
311
for(i=0;i<140;i++){
sum+=A[i]*B[i];
}
STRUCTURE TRANSFORMATION
(b)
312
for(i=0;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i]*B[i];
}
}
for(j=0;j<(140-128);j++,i++){
sum+=A[i]*B[i];
}
SPLITTING
(c) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
313
for(i=0;i<128;){
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
}
for(j=0;j<(140-128);j++,i++){
sum+=A[i]*B[i];
}
INSTRUCTION INSERTION
(d) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
314
for(i=0;i<128;){
dpref(&A[i+32]);
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
dpref(&B[i+32]);
for(j=0;j<16;j++,i++){
sum+=A[i]*B[i];
}
}
for(j=0;j<(140-128);j++,i++){
sum+=A[i]*B[i];
}
PEELING FOLDING
(e)
315
for(i=0;i<140;){
if(i>=128)N=140-128;
else N=32;
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}
SPLITTING
(f) CONDITION TYPE (SIZE REDUCTION)
316
for(i=0,K=0;i<140;){
if(i>=128)N=140-128;
else if(K==0){
K=1,N=16;
} else if(K==1){
K=0,N=16;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}
INSTRUCTION INSERTION
(g) CONDITION TYPE (SIZE REDUCTION)
317
for(i=0,K=0;i<140;){
if(i>=128)N=140-128;
else if(K==0){
dpref(&A[i+32]);
K=1,N=16;
} else if(K==1){
dpref(&B[i+32]);
K=0,N=16;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}

FIG. 19

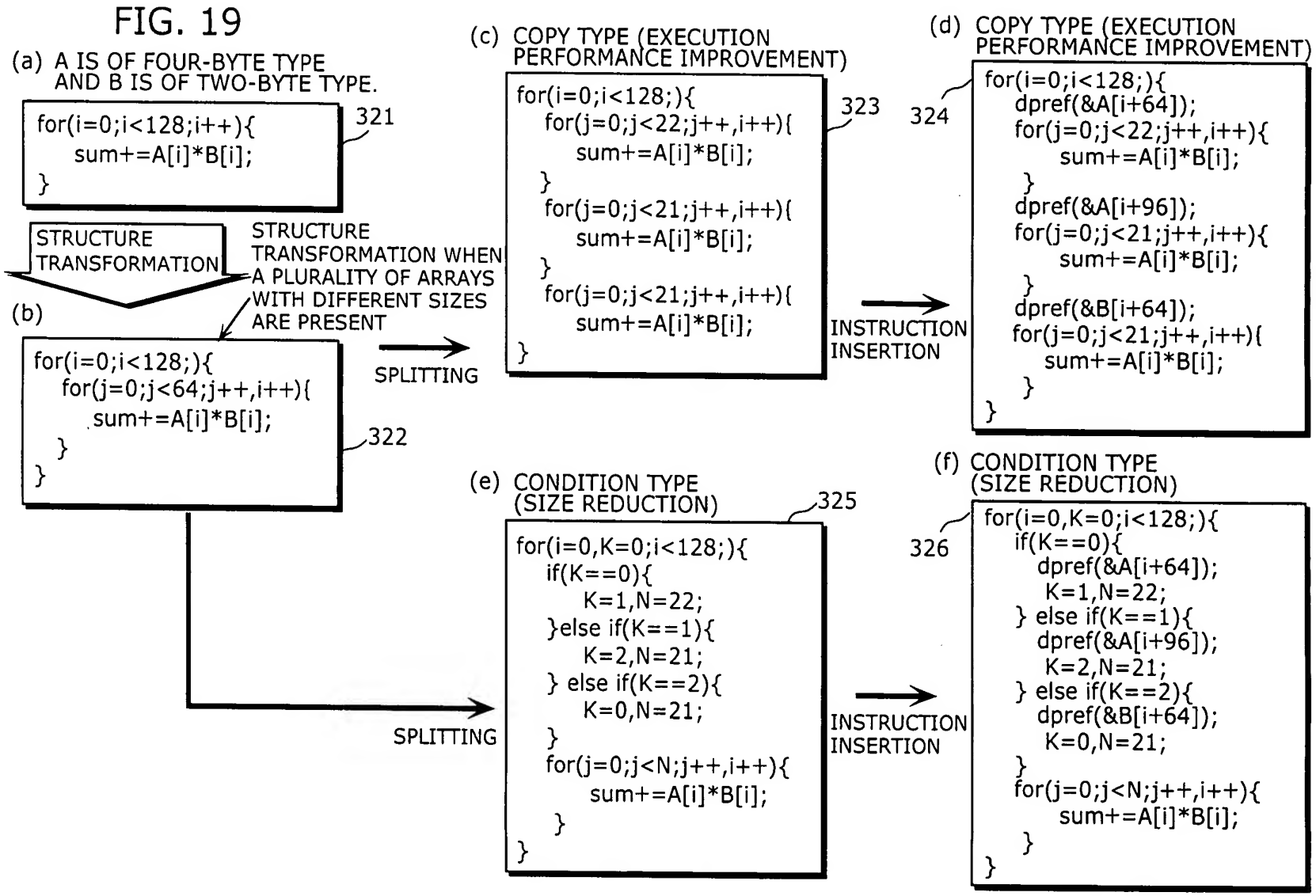

(a) A IS OF FOUR-BYTE TYPE AND B IS OF TWO-BYTE TYPE.
for(i=0;i<128;i++){
sum+=A[i]*B[i];
}
321
STRUCTURE TRANSFORMATION
STRUCTURE TRANSFORMATION WHEN A PLURALITY OF ARRAYS WITH DIFFERENT SIZES ARE PRESENT
(b)
for(i=0;i<128;){
for(j=0;j<64;j++,i++){
sum+=A[i]*B[i];
}
}
322
SPLITTING
(c) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
for(i=0;i<128;){
for(j=0;j<22;j++,i++){
sum+=A[i]*B[i];
}
for(j=0;j<21;j++,i++){
sum+=A[i]*B[i];
}
for(j=0;j<21;j++,i++){
sum+=A[i]*B[i];
}
323
INSTRUCTION INSERTION
(d) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
324
for(i=0;i<128;){
dpref(&A[i+64]);
for(j=0;j<22;j++,i++){
sum+=A[i]*B[i];
}
dpref(&A[i+96]);
for(j=0;j<21;j++,i++){
sum+=A[i]*B[i];
}
dpref(&B[i+64]);
for(j=0;j<21;j++,i++){
sum+=A[i]*B[i];
}
}
SPLITTING
(e) CONDITION TYPE (SIZE REDUCTION)
325
for(i=0,K=0;i<128;){
if(K==0){
K=1,N=22;
}else if(K==1){
K=2,N=21;
} else if(K==2){
K=0,N=21;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}
INSTRUCTION INSERTION
(f) CONDITION TYPE (SIZE REDUCTION)
326
for(i=0,K=0;i<128;){
if(K==0){
dpref(&A[i+64]);
K=1,N=22;
} else if(K==1){
dpref(&A[i+96]);
K=2,N=21;
} else if(K==2){
dpref(&B[i+64]);
K=0,N=21;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}

FIG. 20

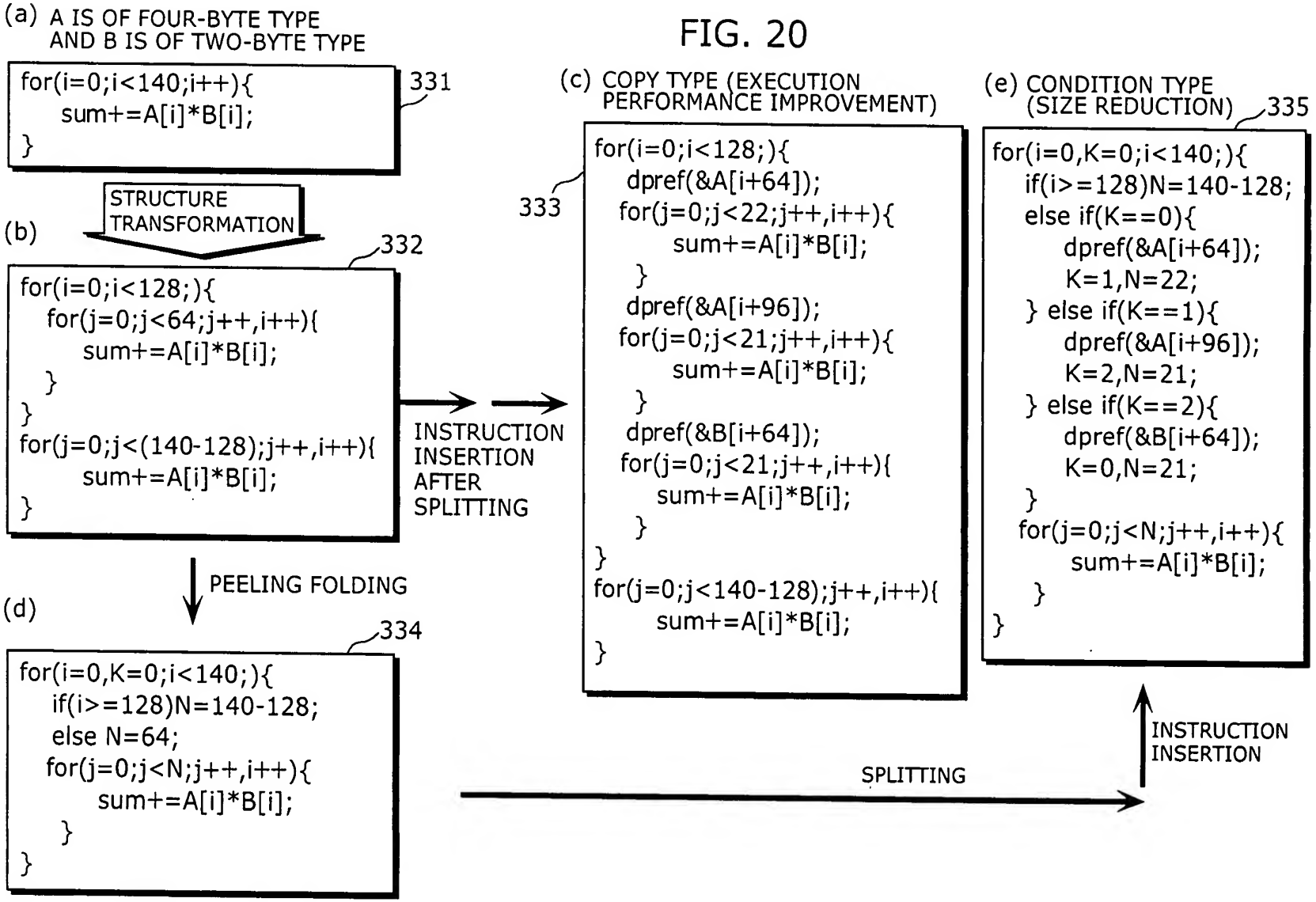

(a) A IS OF FOUR-BYTE TYPE AND B IS OF TWO-BYTE TYPE
for(i=0;i<140;i++){
sum+=A[i]*B[i];
}
331
STRUCTURE TRANSFORMATION
(b)
332
for(i=0;i<128;){
for(j=0;j<64;j++,i++){
sum+=A[i]*B[i];
}
}
for(j=0;j<(140-128);j++,i++){
sum+=A[i]*B[i];
}
INSTRUCTION INSERTION AFTER SPLITTING
(c) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
333
for(i=0;i<128;){
dpref(&A[i+64]);
for(j=0;j<22;j++,i++){
sum+=A[i]*B[i];
}
dpref(&A[i+96]);
for(j=0;j<21;j++,i++){
sum+=A[i]*B[i];
}
dpref(&B[i+64]);
for(j=0;j<21;j++,i++){
sum+=A[i]*B[i];
}
}
for(j=0;j<140-128);j++,i++){
sum+=A[i]*B[i];
}
(e) CONDITION TYPE (SIZE REDUCTION)
335
for(i=0,K=0;i<140;){
if(i>=128)N=140-128;
else if(K==0){
dpref(&A[i+64]);
K=1,N=22;
} else if(K==1){
dpref(&A[i+96]);
K=2,N=21;
} else if(K==2){
dpref(&B[i+64]);
K=0,N=21;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}
PEELING FOLDING
(d)
334
for(i=0,K=0;i<140;){
if(i>=128)N=140-128;
else N=64;
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i];
}
}
SPLITTING
INSTRUCTION INSERTION

FIG. 21

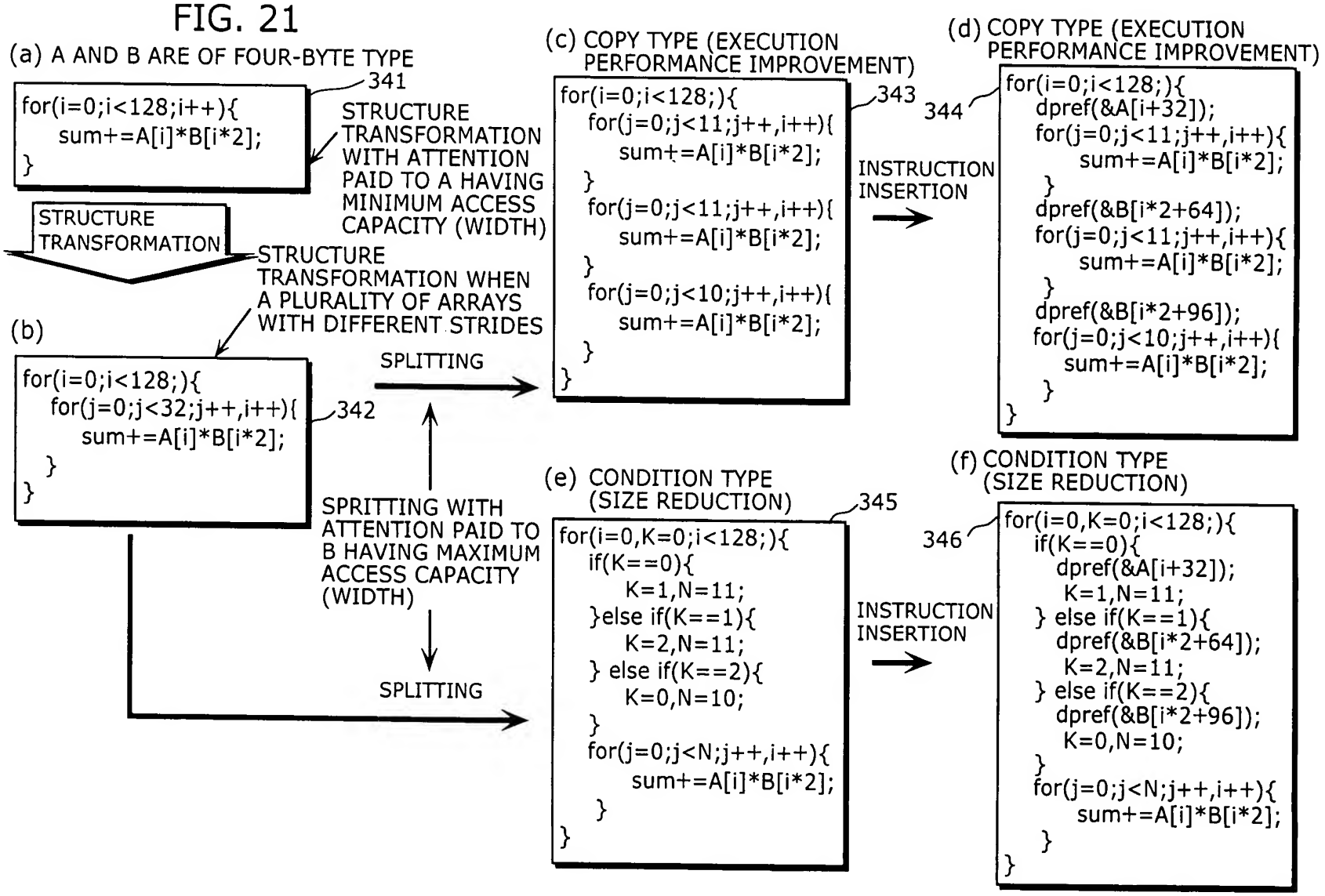

(a) A AND B ARE OF FOUR-BYTE TYPE
for(i=0;i<128;i++){
sum+=A[i]*B[i*2];
}
341
STRUCTURE TRANSFORMATION WITH ATTENTION PAID TO A HAVING MINIMUM ACCESS CAPACITY (WIDTH)
STRUCTURE TRANSFORMATION
STRUCTURE TRANSFORMATION WHEN A PLURALITY OF ARRAYS WITH DIFFERENT STRIDES
(b)
for(i=0;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i]*B[i*2];
}
}
342
SPLITTING
SPRITTING WITH ATTENTION PAID TO B HAVING MAXIMUM ACCESS CAPACITY (WIDTH)
SPLITTING
(c) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
for(i=0;i<128;){
for(j=0;j<11;j++,i++){
sum+=A[i]*B[i*2];
}
for(j=0;j<11;j++,i++){
sum+=A[i]*B[i*2];
}
for(j=0;j<10;j++,i++){
sum+=A[i]*B[i*2];
}
}
343
INSTRUCTION INSERTION
(d) COPY TYPE (EXECUTION PERFORMANCE IMPROVEMENT)
344
for(i=0;i<128;){
dpref(&A[i+32]);
for(j=0;j<11;j++,i++){
sum+=A[i]*B[i*2];
}
dpref(&B[i*2+64]);
for(j=0;j<11;j++,i++){
sum+=A[i]*B[i*2];
}
dpref(&B[i*2+96]);
for(j=0;j<10;j++,i++){
sum+=A[i]*B[i*2];
}
}
(e) CONDITION TYPE (SIZE REDUCTION)
for(i=0,K=0;i<128;){
if(K==0){
K=1,N=11;
}else if(K==1){
K=2,N=11;
} else if(K==2){
K=0,N=10;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i*2];
}
}
345
INSTRUCTION INSERTION
(f) CONDITION TYPE (SIZE REDUCTION)
346
for(i=0,K=0;i<128;){
if(K==0){
dpref(&A[i+32]);
K=1,N=11;
} else if(K==1){
dpref(&B[i*2+64]);
K=2,N=11;
} else if(K==2){
dpref(&B[i*2+96]);
K=0,N=10;
}
for(j=0;j<N;j++,i++){
sum+=A[i]*B[i*2];
}
}

FIG. 22

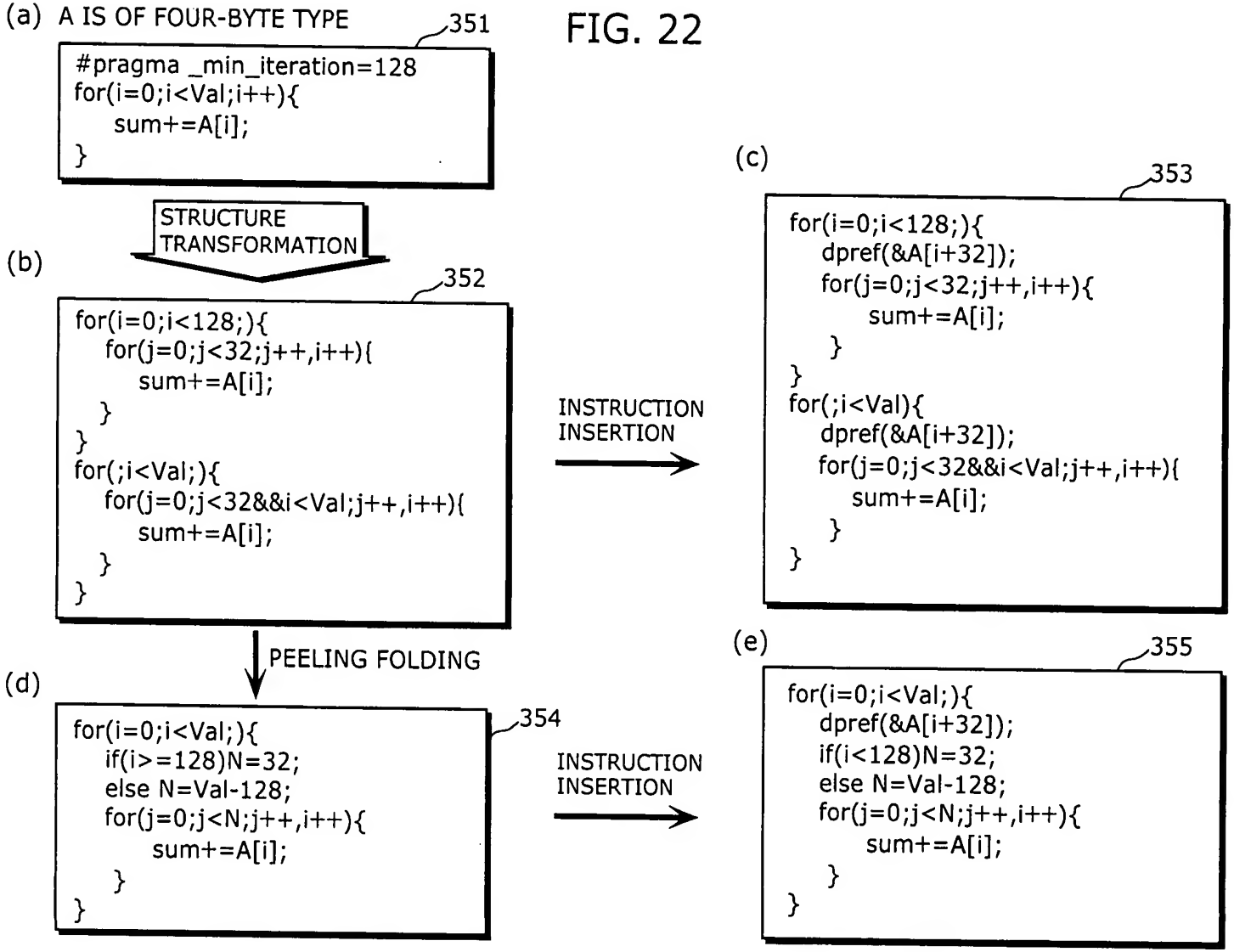

(a) A IS OF FOUR-BYTE TYPE
351
#pragma _min_iteration=128
for(i=0;i<Val;i++){
sum+=A[i];
}
STRUCTURE TRANSFORMATION
(b)
352
for(i=0;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
for(;i<Val;){
for(j=0;j<32&&i<Val;j++,i++){
sum+=A[i];
}
}
INSTRUCTION INSERTION
(c)
353
for(i=0;i<128;){
dpref(&A[i+32]);
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
for(;i<Val){
dpref(&A[i+32]);
for(j=0;j<32&&i<Val;j++,i++){
sum+=A[i];
}
}
PEELING FOLDING
(d)
354
for(i=0;i<Val;){
if(i>=128)N=32;
else N=Val-128;
for(j=0;j<N;j++,i++){
sum+=A[i];
}
}
INSTRUCTION INSERTION
(e)
355
for(i=0;i<Val;){
dpref(&A[i+32]);
if(i<128)N=32;
else N=Val-128;
for(j=0;j<N;j++,i++){
sum+=A[i];
}
}

FIG. 23

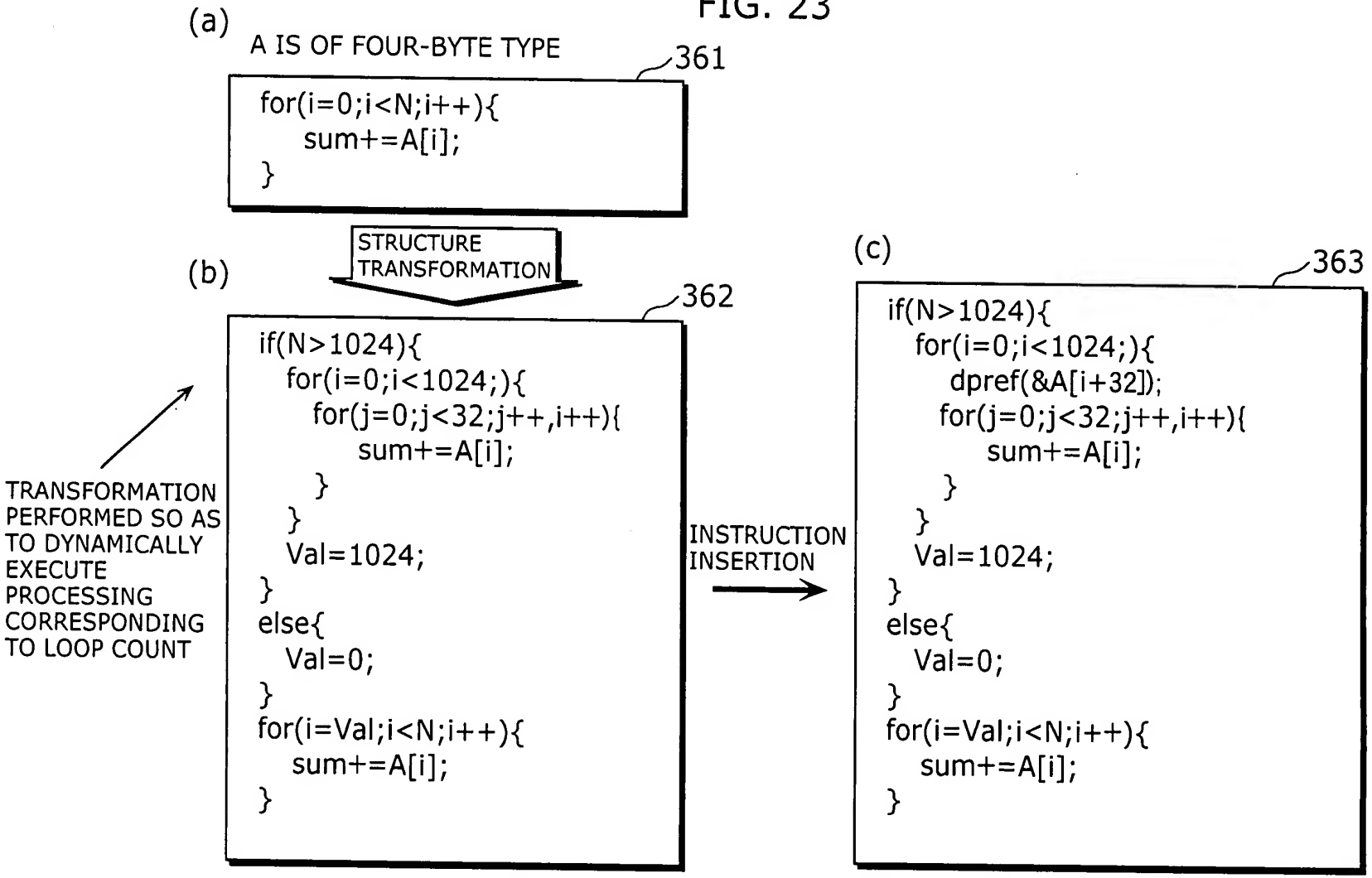

(a)
A IS OF FOUR-BYTE TYPE
361
for(i=0;i<N;i++){
sum+=A[i];
}
STRUCTURE TRANSFORMATION
(b)
362
if(N>1024){
for(i=0;i<1024;){
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
Val=1024;
}
else{
Val=0;
}
for(i=Val;i<N;i++){
sum+=A[i];
}
TRANSFORMATION PERFORMED SO AS TO DYNAMICALLY EXECUTE PROCESSING CORRESPONDING TO LOOP COUNT
INSTRUCTION INSERTION
(c)
363
if(N>1024){
for(i=0;i<1024;){
dpref(&A[i+32]);
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
Val=1024;
}
else{
Val=0;
}
for(i=Val;i<N;i++){
sum+=A[i];
}

FIG. 24

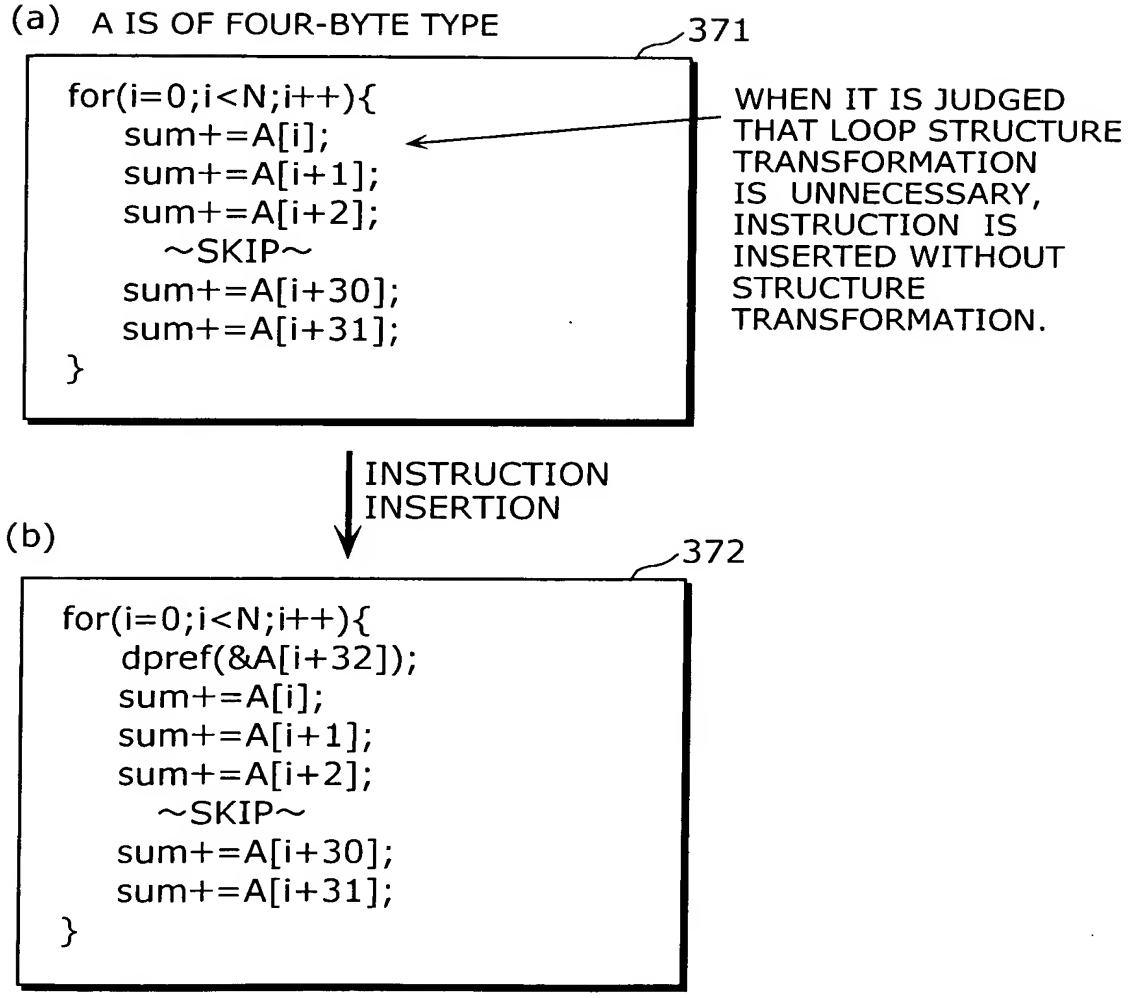

(a) A IS OF FOUR-BYTE TYPE
371
for(i=0;i<N;i++){
sum+=A[i];
sum+=A[i+1];
sum+=A[i+2];
~SKIP~
sum+=A[i+30];
sum+=A[i+31];
}
WHEN IT IS JUDGED THAT LOOP STRUCTURE TRANSFORMATION IS UNNECESSARY, INSTRUCTION IS INSERTED WITHOUT STRUCTURE TRANSFORMATION.
INSTRUCTION INSERTION
(b)
372
for(i=0;i<N;i++){
dpref(&A[i+32]);
sum+=A[i];
sum+=A[i+1];
sum+=A[i+2];
~SKIP~
sum+=A[i+30];
sum+=A[i+31];
}

FIG. 25

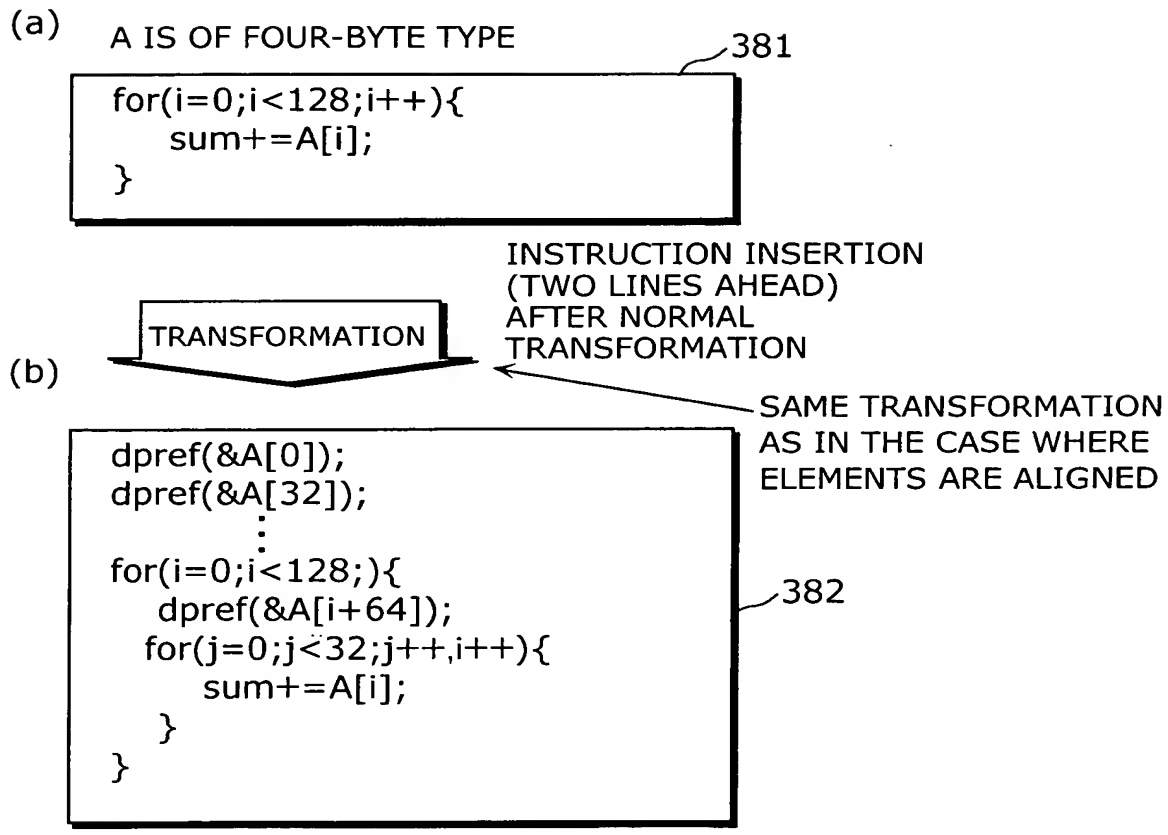

(a)
A IS OF FOUR-BYTE TYPE
381
for(i=0;i<128;i++){
sum+=A[i];
}
INSTRUCTION INSERTION
(TWO LINES AHEAD)
AFTER NORMAL
TRANSFORMATION
TRANSFORMATION
(b)
SAME TRANSFORMATION
AS IN THE CASE WHERE
ELEMENTS ARE ALIGNED
dpref(&A[0]);
dpref(&A[32]);
for(i=0;i<128;){
dpref(&A[i+64]);
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
382

FIG. 26

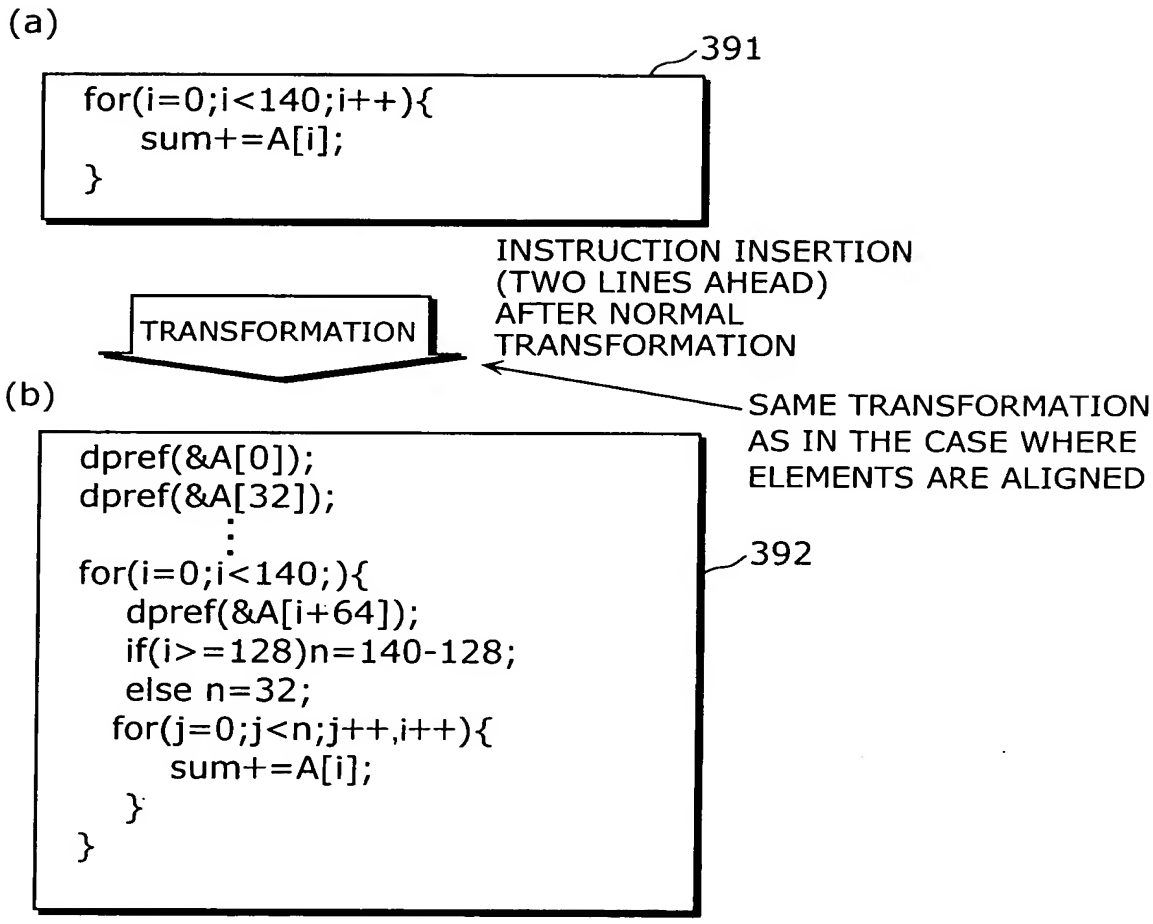

(a)
391
for(i=0;i<140;i++){
sum+=A[i];
}
TRANSFORMATION
INSTRUCTION INSERTION
(TWO LINES AHEAD)
AFTER NORMAL
TRANSFORMATION
SAME TRANSFORMATION
AS IN THE CASE WHERE
ELEMENTS ARE ALIGNED
(b)
dpref(&A[0]);
dpref(&A[32]);
for(i=0;i<140;){
dpref(&A[i+64]);
if(i>=128)n=140-128;
else n=32;
for(j=0;j<n;j++,i++){
sum+=A[i];
}
}
392

FIG. 27

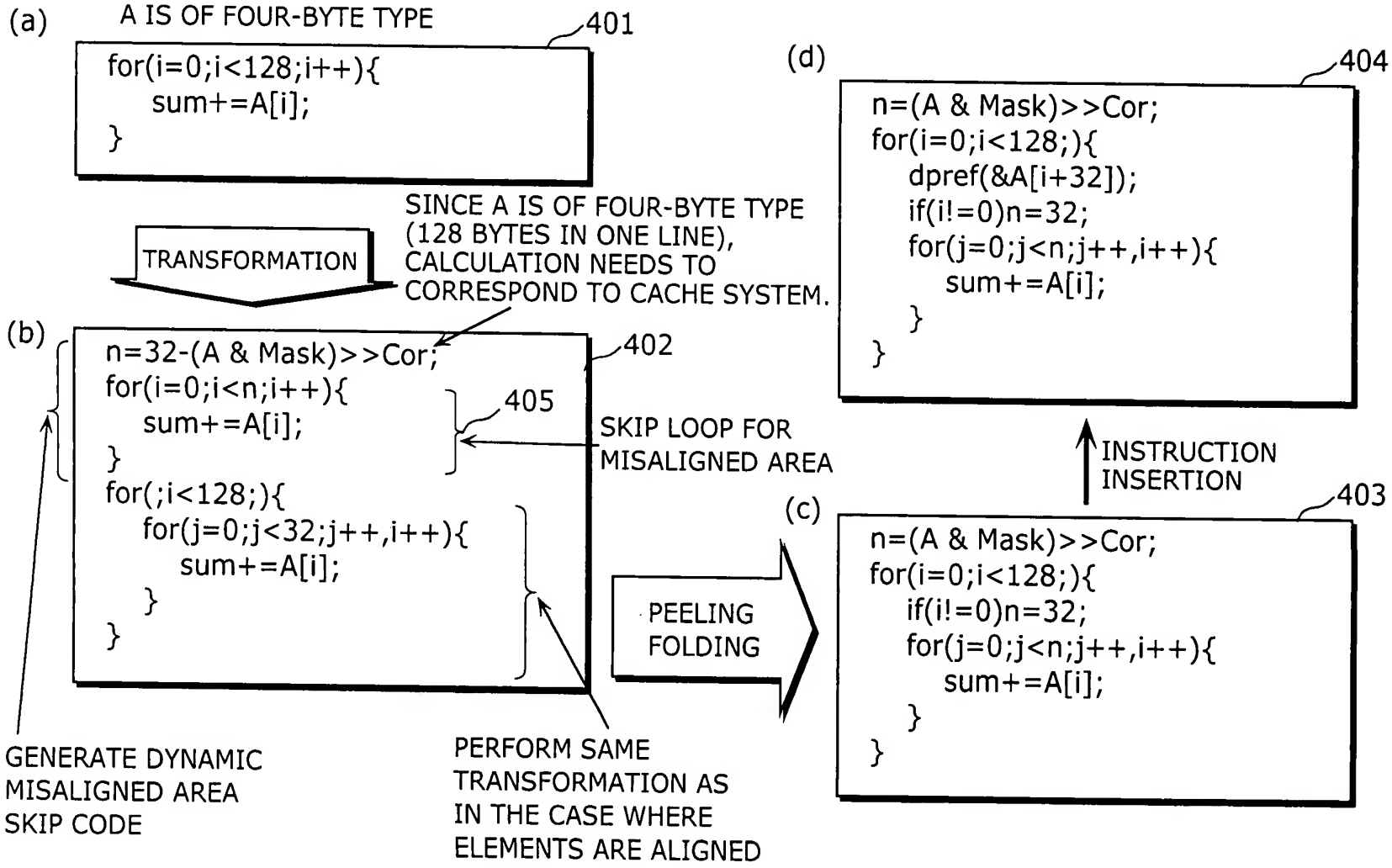

(a)
A IS OF FOUR-BYTE TYPE
401
for(i=0;i<128;i++){
sum+=A[i];
}
TRANSFORMATION
SINCE A IS OF FOUR-BYTE TYPE (128 BYTES IN ONE LINE), CALCULATION NEEDS TO CORRESPOND TO CACHE SYSTEM.
(b)
n=32-(A & Mask)>>Cor;
for(i=0;i<n;i++){
sum+=A[i];
}
for(;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
402
405
SKIP LOOP FOR MISALIGNED AREA
GENERATE DYNAMIC MISALIGNED AREA SKIP CODE
PERFORM SAME TRANSFORMATION AS IN THE CASE WHERE ELEMENTS ARE ALIGNED
PEELING FOLDING
(c)
n=(A & Mask)>>Cor;
for(i=0;i<128;){
if(i!=0)n=32;
for(j=0;j<n;j++,i++){
sum+=A[i];
}
}
403
INSTRUCTION INSERTION
(d)
n=(A & Mask)>>Cor;
for(i=0;i<128;){
dpref(&A[i+32]);
if(i!=0)n=32;
for(j=0;j<n;j++,i++){
sum+=A[i];
}
}
404

FIG. 28

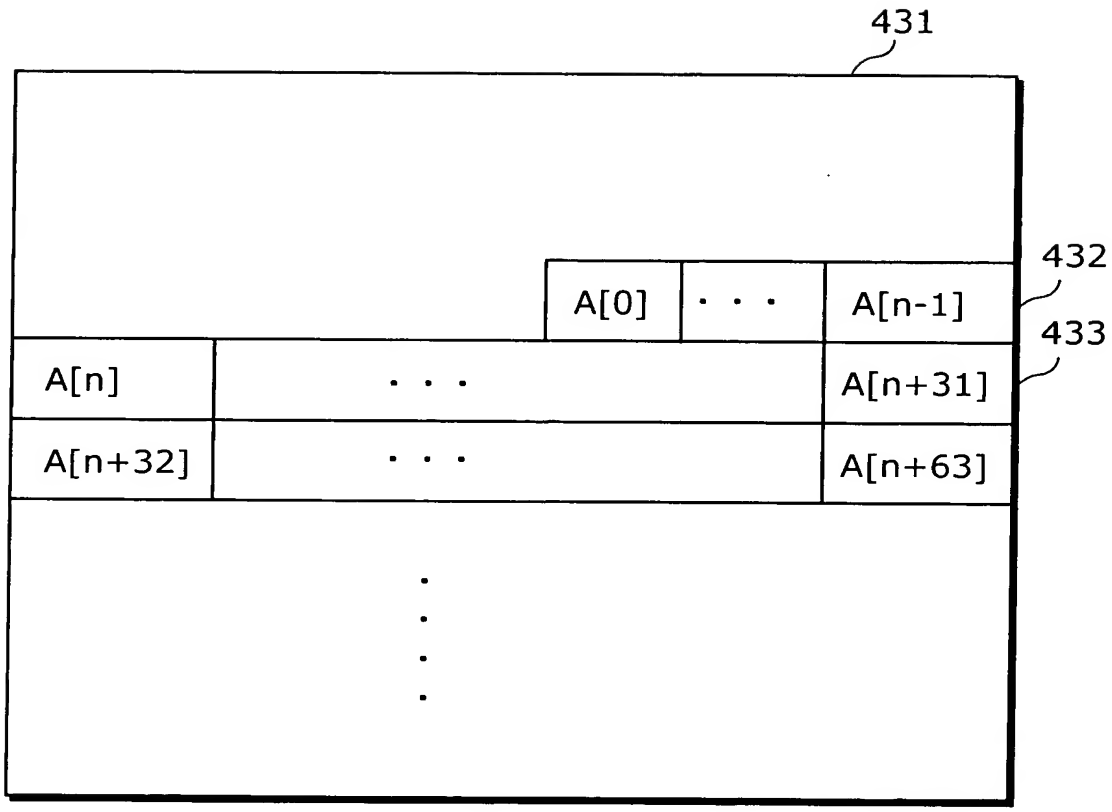


FIG. 29

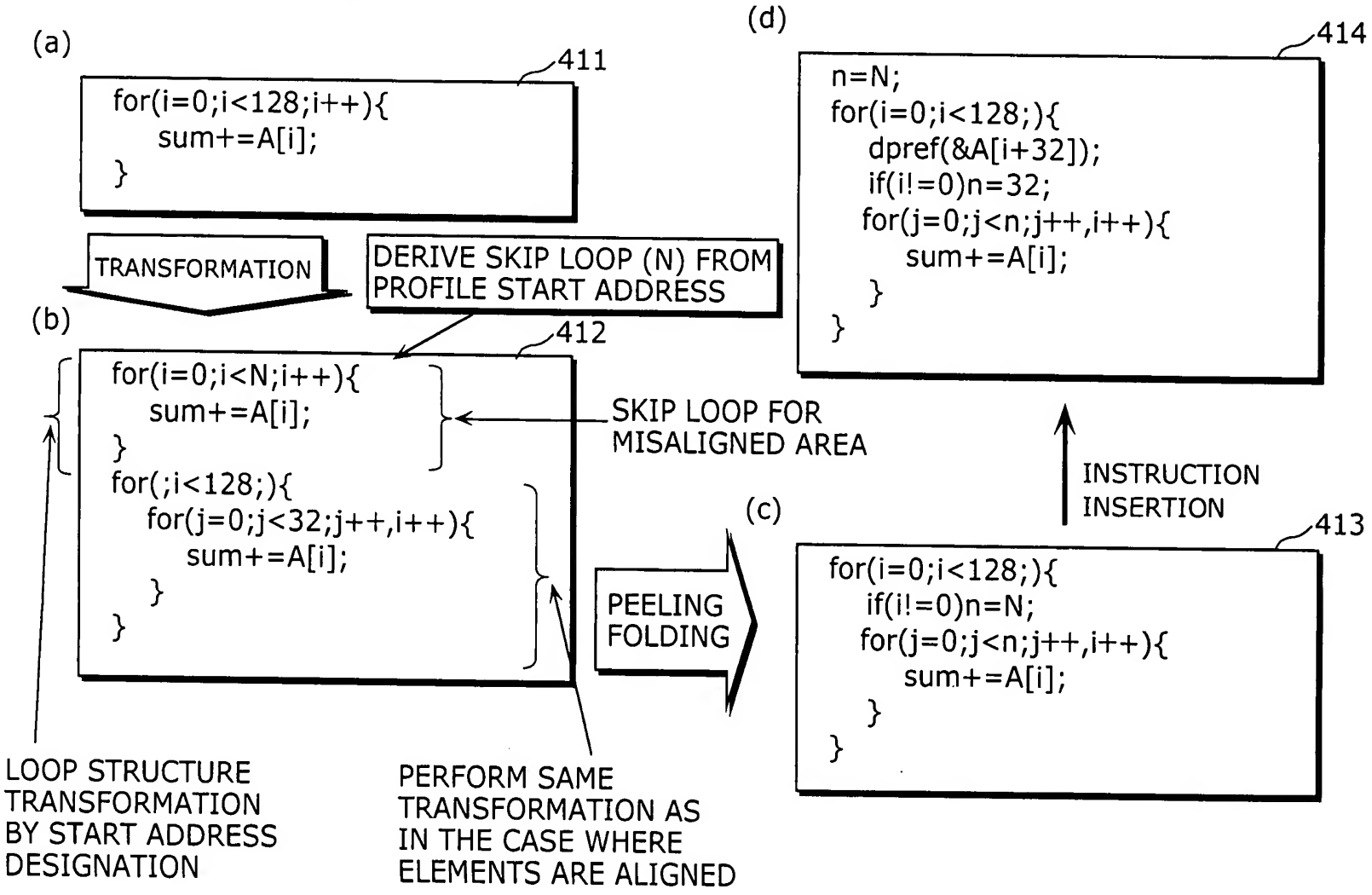

(a)
411
for(i=0;i<128;i++){
sum+=A[i];
}
TRANSFORMATION
DERIVE SKIP LOOP (N) FROM PROFILE START ADDRESS
(b)
412
for(i=0;i<N;i++){
sum+=A[i];
}
for(;i<128;){
for(j=0;j<32;j++,i++){
sum+=A[i];
}
}
SKIP LOOP FOR MISALIGNED AREA
PEELING FOLDING
(c)
413
for(i=0;i<128;){
if(i!=0)n=N;
for(j=0;j<n;j++,i++){
sum+=A[i];
}
}
INSTRUCTION INSERTION
(d)
414
n=N;
for(i=0;i<128;){
dpref(&A[i+32]);
if(i!=0)n=32;
for(j=0;j<n;j++,i++){
sum+=A[i];
}
}
LOOP STRUCTURE TRANSFORMATION BY START ADDRESS DESIGNATION
PERFORM SAME TRANSFORMATION AS IN THE CASE WHERE ELEMENTS ARE ALIGNED

FIG. 30

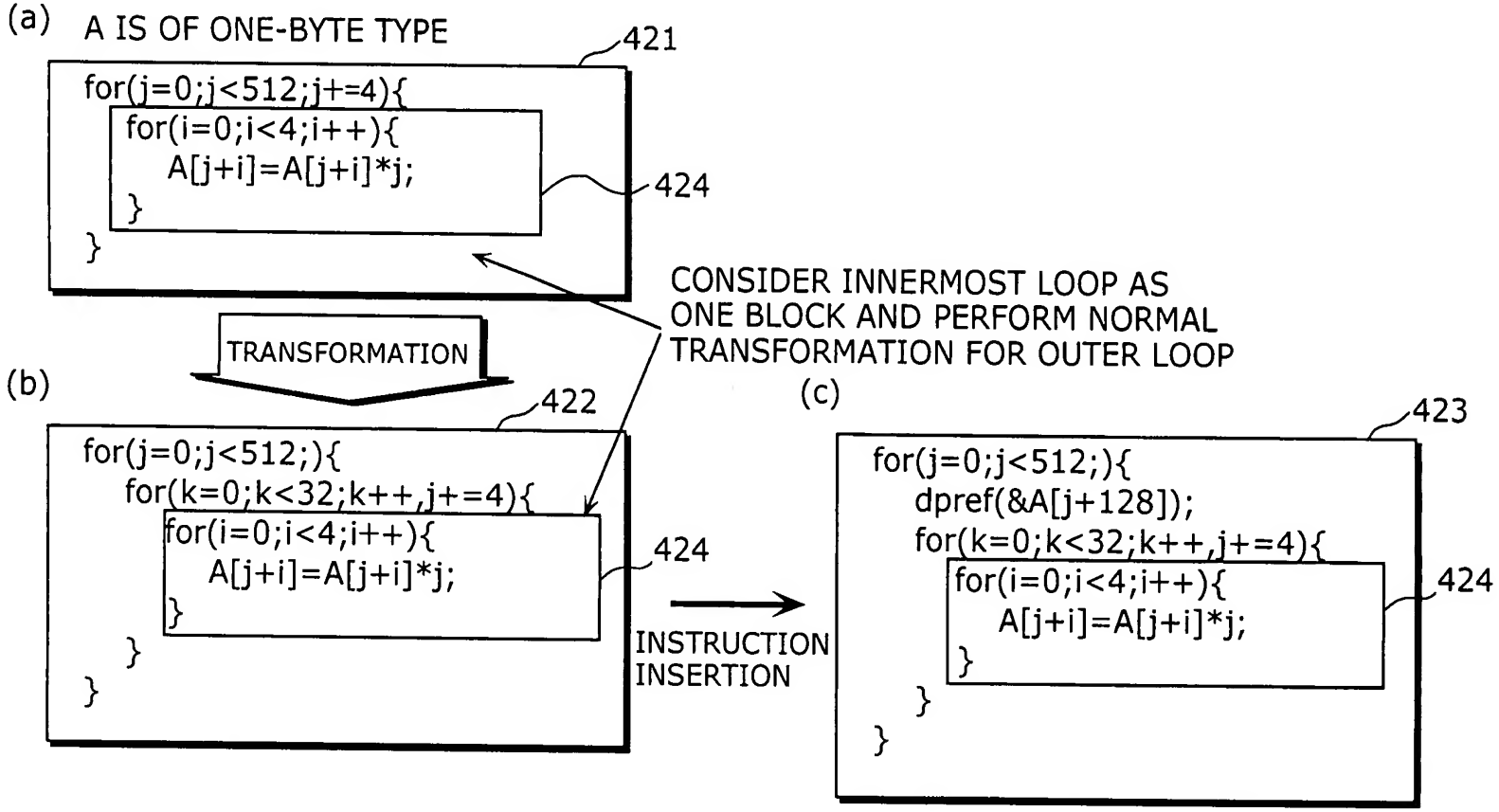

(a)
A IS OF ONE-BYTE TYPE
421
for(j=0;j<512;j+=4){
for(i=0;i<4;i++){
A[j+i]=A[j+i]*j;
}
}
424
CONSIDER INNERMOST LOOP AS ONE BLOCK AND PERFORM NORMAL TRANSFORMATION FOR OUTER LOOP
TRANSFORMATION
(b)
422
for(j=0;j<512;){
for(k=0;k<32;k++,j+=4){
for(i=0;i<4;i++){
A[j+i]=A[j+i]*j;
}
}
}
424
INSTRUCTION INSERTION
(c)
423
for(j=0;j<512;){
dpref(&A[j+128]);
for(k=0;k<32;k++,j+=4){
for(i=0;i<4;i++){
A[j+i]=A[j+i]*j;
}
}
}
424

FIG. 31

(a)

```
int b[128]:
#pragma _loop_tiling_dpref b
for (i=0; i<128; i++)
{
        a[i] = b[i];
}
```

(b)

```
for (i=0; i<128; )
{
        dpref(&b[i+32]);
        for (j =0; j<32; j++, i++){
                a[i] = b[i];
        }
}
```

FIG. 32

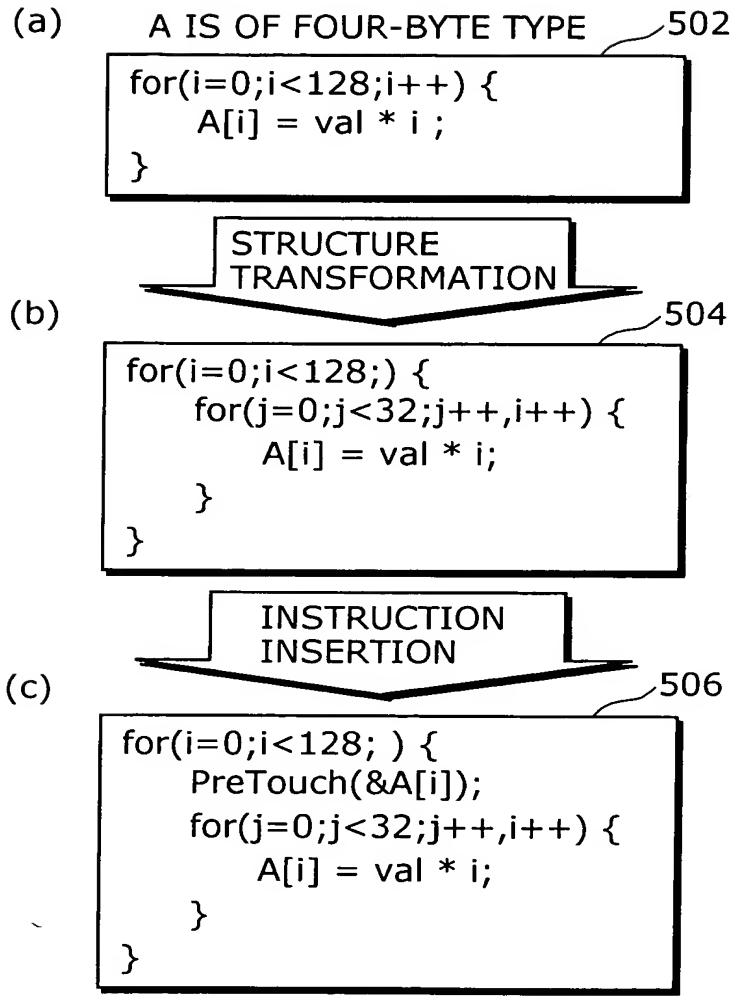

(a)
A IS OF FOUR-BYTE TYPE
502
for(i=0;i<128;i++) {
A[i] = val * i ;
}
STRUCTURE TRANSFORMATION
(b)
504
for(i=0;i<128;) {
for(j=0;j<32;j++,i++) {
A[i] = val * i;
}
}
INSTRUCTION INSERTION
(c)
506
for(i=0;i<128;) {
PreTouch(&A[i]);
for(j=0;j<32;j++,i++) {
A[i] = val * i;
}
}

FIG. 33

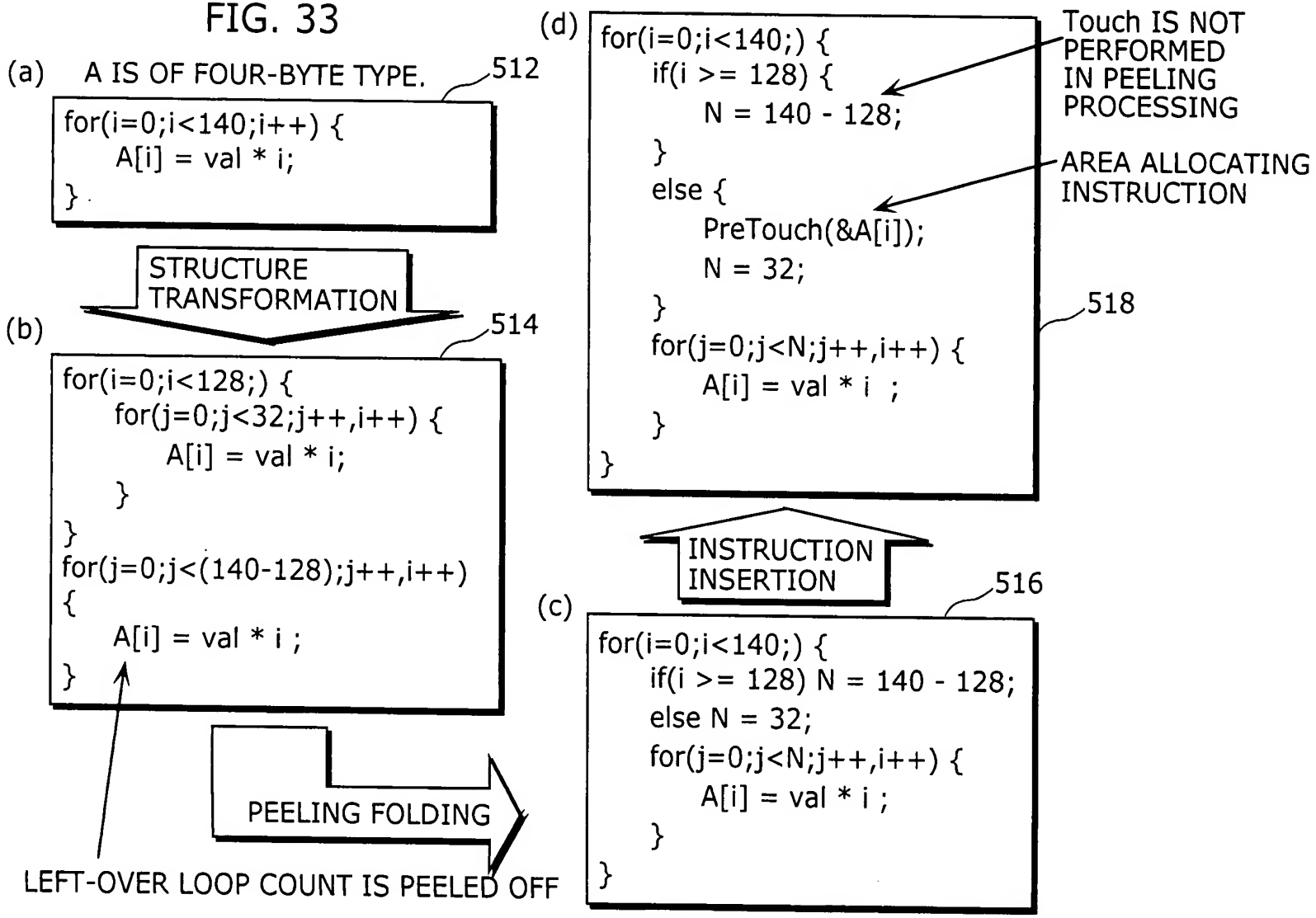
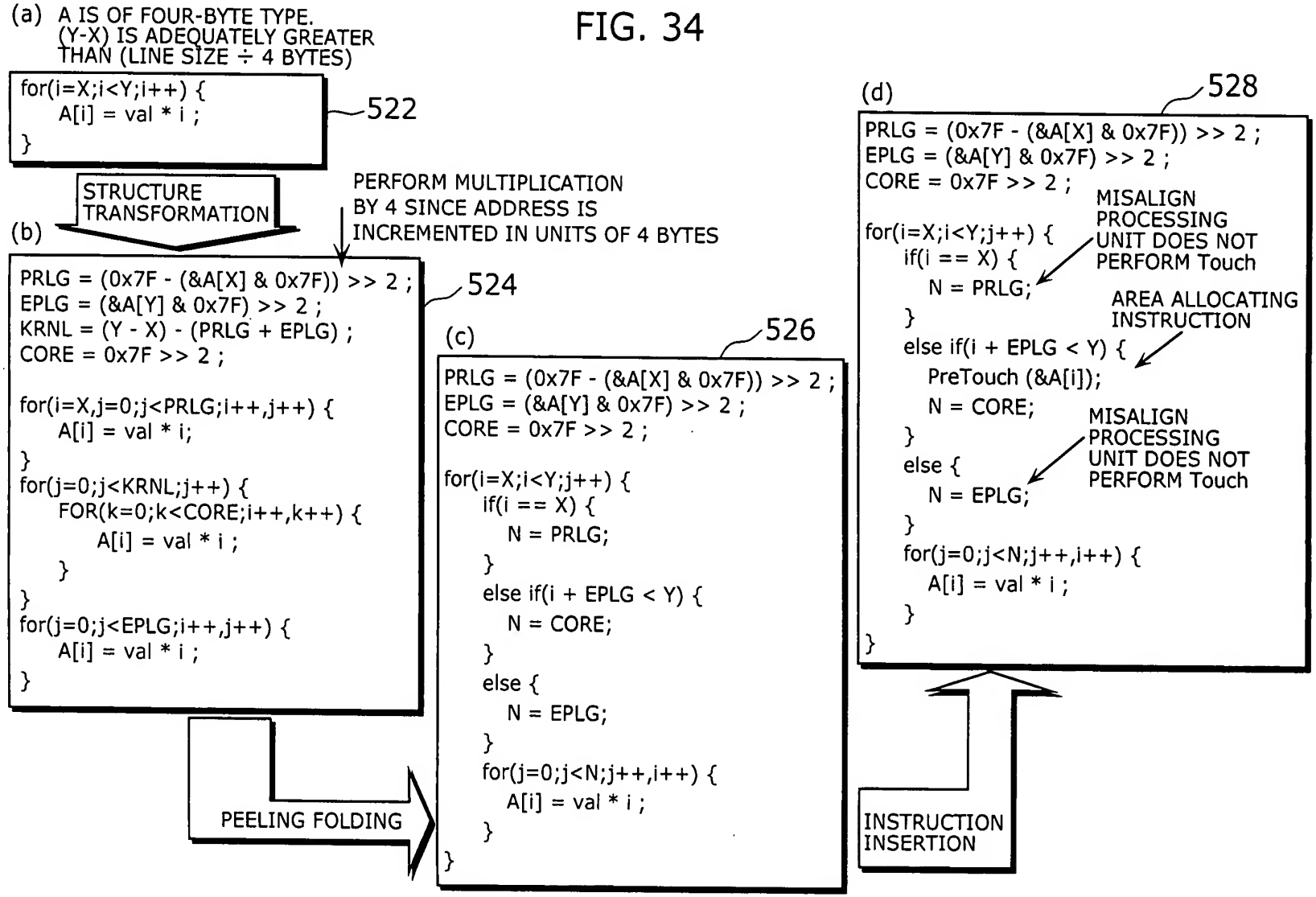

(a) A IS OF FOUR-BYTE TYPE.
512
for(i=0;i<140;i++) {
A[i] = val * i;
}
STRUCTURE TRANSFORMATION
(b)
514
for(i=0;i<128;) {
for(j=0;j<32;j++,i++) {
A[i] = val * i;
}
}
for(j=0;j<(140-128);j++,i++)
{
A[i] = val * i ;
}
PEELING FOLDING
LEFT-OVER LOOP COUNT IS PEELED OFF
(d)
for(i=0;i<140;) {
if(i >= 128) {
N = 140 - 128;
}
else {
PreTouch(&A[i]);
N = 32;
}
for(j=0;j<N;j++,i++) {
A[i] = val * i ;
}
}
Touch IS NOT PERFORMED IN PEELING PROCESSING
AREA ALLOCATING INSTRUCTION
518
INSTRUCTION INSERTION
(c)
516
for(i=0;i<140;) {
if(i >= 128) N = 140 - 128;
else N = 32;
for(j=0;j<N;j++,i++) {
A[i] = val * i ;
}
}

FIG. 34


(a) A IS OF FOUR-BYTE TYPE.
(Y-X) IS ADEQUATELY GREATER
THAN (LINE SIZE ÷ 4 BYTES)
for(i=X;i<Y;i++) {
A[i] = val * i ;
}
522
STRUCTURE
TRANSFORMATION
PERFORM MULTIPLICATION
BY 4 SINCE ADDRESS IS
INCREMENTED IN UNITS OF 4 BYTES
(b)
PRLG = (0x7F - (&A[X] & 0x7F)) >> 2 ;
EPLG = (&A[Y] & 0x7F) >> 2 ;
KRNL = (Y - X) - (PRLG + EPLG) ;
CORE = 0x7F >> 2 ;
for(i=X,j=0;j<PRLG;i++,j++) {
A[i] = val * i;
}
for(j=0;j<KRNL;j++) {
FOR(k=0;k<CORE;i++,k++) {
A[i] = val * i ;
}
}
for(j=0;j<EPLG;i++,j++) {
A[i] = val * i ;
}
524
PEELING FOLDING
(c)
526
PRLG = (0x7F - (&A[X] & 0x7F)) >> 2 ;
EPLG = (&A[Y] & 0x7F) >> 2 ;
CORE = 0x7F >> 2 ;
for(i=X;i<Y;j++) {
if(i == X) {
N = PRLG;
}
else if(i + EPLG < Y) {
N = CORE;
}
else {
N = EPLG;
}
for(j=0;j<N;j++,i++) {
A[i] = val * i ;
}
}
INSTRUCTION
INSERTION
(d)
528
PRLG = (0x7F - (&A[X] & 0x7F)) >> 2 ;
EPLG = (&A[Y] & 0x7F) >> 2 ;
CORE = 0x7F >> 2 ;
for(i=X;i<Y;j++) {
if(i == X) {
N = PRLG;
}
else if(i + EPLG < Y) {
PreTouch (&A[i]);
N = CORE;
}
else {
N = EPLG;
}
for(j=0;j<N;j++,i++) {
A[i] = val * i ;
}
}
MISALIGN
PROCESSING
UNIT DOES NOT
PERFORM Touch
AREA ALLOCATING
INSTRUCTION
MISALIGN
PROCESSING
UNIT DOES NOT
PERFORM Touch